C64-FORTH UPGRADE

C64-FORTH(tm) has been upgraded! It now meets the true 79-standard. Many additional words have been added to provide extended utility for the user. Error handling and byte I/O is improved. C64-FORTH now works with a broader range of IEEE interface cartridges. You may now drop out of FORTH TRACE mode into a machine language monitor, and re-enter easily. TAB and DUPLICATE-A-LINE capability has been added to the EDITOR.

String extensions are provided as source screens on the C64-FORTH disk. All string functions available under Commodore-64 BASIC are accomodated. A few additional functions, such as POS$ (search for a substring within a string), make life all that much easier in FORTH-land.

A full floating-point command set is available in C64-FORTH. Every functon in Commodore-64 BASIC that handles real numbers is duplicated in C64-FORTH. Unlike some f.p. extensions for other FORTHS, nearly all the routines are written in machine code. Floating-point functions are very fast. Routines like FPSTR$ produce an ASCII string representaton of a floating-point number that is compatible with the string-extensions. FPRND provides a random number based upon a seed parameter the same as in BASIC. Integer to f.p. and f.p. to integer functions are provided for both single and double precision, signed and unsigned values.

Full graphics functions are provided. Most graphics functions are machine code subroutines, with FORTH level calls. Line, circle, arc, box, etc. functions are very fast. Program designers looking to use C64-FORTH for commercial products will be able to write entire game programs in C64-FORTH assembler. The graphics routines support full scaling and windowing (clipping within a sub-portion of the display screen). The routines are universal in respect to working in hi-res or med-res multi-color mode. Integer SINE and COSINE routines are provided, as well as a 32x32 bit multiply routine. Relative line modes are supported, as well as standard LINE and LINETO. One command allows you to position the display and color memories anywhere in memory (the pixel-modifying routine automatically maps in and out the Commodore-64 kernal ROM to allow full use of the 8K RAM underneath for graphics). Any character in the C64 character ROM (or in a user-redefined character set) may be mapped onto the hi or med-res bit map memory in any of 16 zoom factors starting at any X,Y location! Full sprite commands available, as well as an easy way to define sprites in FORTH screens. Pixel, line, arc, etc. all work in SET-BIT mode, CLEAR-BIT mode, or XOR-mode. Some graphics demos are provided as source screens.

And last, but not least, the manual has been improved. Many of the previous sections have been re-written. In the interest of beginners, many topics are now covered in greater detail. Much more technical detail is included for more sophisticated users. New sections have been added for string, floating-point, and graphics commands.

C64=FORTH UPGRADE

A few notes are in order for previous C64-FORTH users. This upgrade meets the FORTH=79 standard only. There are a few surface-level changes between fig-std FORTH and 79-std. In 79-std, variables are not pre-initialized. Whereas in the earlier C64-FORTH you created a variable by:

```
-1  VARIABLE  MIN1
```

In 79-std, if you wish to initialize a variable you just created you store a value into it using ! :

```
VARIABLE  MIN1    -1  MIN1  !
```

Remember that 2VARIABLE will also be the same!

```
2VARIABLE  DMIN1   -1.  DMIN1  2!
```

WORD under the 79-std also works a little differently. It now returns the address of where the string is returned, which will be the current value of HERE. Fig-std WORD did not return any address.

<BUILDS is no longer available. Instead CREATE must be used with DOES>.

These are the only changes that are immediately obvious and will require you to modify some of your source code. Internally, the operation of words like DO..LOOP may work a little differently. I suggest you read the section in the new manual on DO..LOOPs.

A few errors are obvious in the new manual and should be corrected. On page D-7, FPRND states an integer should be passed to the routine. This should be a floating-point value! The stack then will be: f1 --- fprnd

Also, on page C-6, the description of MCSETVAL says the 2-bit value to write to multicolor memory when in clear-bit mode should be changed to read "the 2-bit value to write to multicolor memory when in SET-BIT mode".

On page ? the description to load C64-FORTH should read LOAD"C64-FORTH",8.

C64-FORTH™

A FORTH LANGUAGE SYSTEM FOR THE COMMODORE-64

by Gregg Harris

C64-FORTH CONFORMS TO:

FORTH-79 STANDARD WITH DOUBLE-NUMBER EXTENSIONS

A product of:

Performance Micro Products
770 Dedham St.
Canton, MA 02021 USA

Any mention within this manual of COMMODORE 64, C64, or 'the 64', is reference to the COMMODORE 64 personal computer manufactured and marketed by COMMODORE BUSINESS MACHINES, INC. The FORTH language standards are defined and maintained by the FORTH INTEREST GROUP (P.O. Box 1105, San Carlos, CA. 94070). This version of C64-FORTH meets the FORTH-79 standard with Double-Number Extensions.

We would like to acknowledge Mr. Charles H. Moore, the creater of FORTH, and all of the contributors of FORTH INC. who have helped make FORTH something more than a word to stick after 'GO'. Some of the descriptions in our ASSEMBLER section reflect the work of Bill Ragsdale, from his treatise on a Forth Assembler for the 6502. The method adopted for scaling in the graphics routines was taken from work by Allen Tracht. Our thanks to Allen for providing a practical implementation.

We believe C64-FORTH to be as close to the FORTH-79 standard as is possible in the word set provided, the way standard words work as generally accepted, and in the manual's attempt to convey the operation of those words. Upon shipment of the first copy, there were no known flaws in achieving the above goal, nor 'bugs' within any of the code. If any user of C64-FORTH finds otherwise, we emphatically encourge notification of such. We would not want buy software from a company that did not care to provide support once the product's out the door, so we're not going to ask you to.

PERFORMANCE MICRO PRODUCTS, INC.
770 Dedham St. -S2
Canton, Mass. 02021

C64-FORTH™

COPYRIGHT 1983 BY PERFORMANCE MICRO PRODUCTS INC.
ALL RIGHTS RESERVED

- COPYRIGHT NOTICE

The C64-FORTH package including the object code, and FORTH source screens, both supplied on disk or cassette, and this user manual are serial numbered and are protected by a copyright. Reproduction by any method whatsoever of these copyrighted programs or the user manual is expressly prohibited without the expressed written permission of Performance Micro Products, Inc. To protect your copy of this software, you are permitted to make one backup copy of the contents of this disk or cassette. This backup may not be redistributed.

Except for a single backup copy, it is a Federal crime to make a copy of the manual, floppy disk, or cassette for use by anyone other than the individual who purchased this software, or the individual for which a company purchased this software. A reward will be provided for information which leads to the prosecution and conviction of parties who violate this copyright.

With respect to distribution of application programs created using C64-FORTH, we allow distribution of this software using the "SAVETURNKEY" command with no legal restrictions or license fee, providing that the end user does not have access to the FORTH words contained in the C64-FORTH nucleus. Programs created with "SAVETURNKEY" will operate in a stand-alone configuration and will not require the purchase of a C64-FORTH.

The individual, for whom this software was purchased, may save new or modified versions of C64-FORTH, for his/her own use only, onto disk or cassette by using the "SAVESYSTEM" command under C64-FORTH. Since any program saved under this command is a fully usable FORTH system, the above restrictions on redistribution also apply. Disks or cassettes containing ONLY FORTH source screens, and not any usable copies of any FORTH system generated from C64-FORTH, may be freely distributed.

Serial numbers are embedded in several different formats in several different places in the C64-FORTH software. We intend to identify and prosecute the source of any original or modified copy of C64-FORTH, not generated with the SAVETURNKEY command, that was redistributed for compensation or not.

This policy is in the best interests of the legitimate purchasers and users of C64-FORTH. Performance Micro Products would like to continue to offer a broad spectrum of software products including many extension and applications packages for C64-FORTH. We cannot afford the time and financial demands required to produce very versatile, user-forgiving, and extensively tested software if users of C64-FORTH do not respect our right to compensation for our investment.

- LIMITED WARRANTY INFORMATION

For a period of ninety (90) days from the date of sales, Performance Micro Products, Inc. warrants that the cassette or diskette containing software programs is free from defects. In the event of a defect the customer's sole and exclusive remedy is limited to the correction of the defect by replacement or complete refund at Performance Micro Products election and sole expense. Performance Micro Products has no obligation to replace items that have failed due to abuse or misuse. Peformance Micro Products makes no warranty as to the design, capability, capacity, or suitability for use of the software. Software is licensed to the original retail purchaser on an "as is" basis, without warranty. The customers exclusive remedy in the event of a software defect is the replacement or update of the software within 90 days of the date of purchase upon the return of the original cassette or diskette to Performance Micro Products.

Except as provided herein, Performance Micro Products shall not have any liability with respect to any loss liability or damage caused directly or indirectly by the software sold or furnished by Performance Micro Products. The exclusion of liability shall include but not be limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use of the software.

ABOUT THIS MANUAL

It is not within the scope of this manual to provide a complete introduction to FORTH. There are several good books available that break a new user in on FORTH a little more slowly. The goals of this manual then were defined as:

1) The manual is to serve as a concise reference guide to the words available under, and the architecture of, this C64-FORTH implementation of the FORTH language.

2) The user is assumed to be at least somewhat familiar with the Commodore-64, and has tried writing some programs in BASIC. Equivalent routines in BASIC will be shown whenever a FORTH word or function is similar to one in BASIC.

3) Elaboration on a subject is done to provide enough information so a new user shouldn't need a book on FORTH as long the user is willing to experiment. Elaboration is also done on subjects not usually covered in most books on FORTH.

DON'T BE AFRAID TO EXPERIMENT! Get in there and peek and poke around (or should I say @ and ! around). Look at what you are doing. Take note of conditions before you try something; note results after you try it. Set up conditions to see how different FORTH words work. FORTH can be a lot more rewarding and versatile than languages that 'do it all' for you.

CONVENTIONS IN THIS MANUAL

Any mention of 'the 64', or C64 will of course refer to the Commodore-64, such as "press the STOP key on the 64".

Any input of commands must always be followed by pressing the RETURN key. Nothing will happen until you do. At first, this is indicated in the examples by the symbol <return> as a reminder, but later in the manual it is assumed that you will do it automatically after typing in any commands or numbers.

There are cases where a symbol is used to indicate a string of characters representing a name or something is to be substituted. This will usually be expressed as a keyword within greater-than and less-than symbols. An example is when the symbol ' <name> ' appears, this is an indication that a character string is to be entered, which will be used as the name of a definition being created. If <name> is used later in the description, the SAME character string is to be substituted again.

Examples will have underlined the part typed in by the user ONLY when the response from C64-FORTH is significant. C64-FORTH responds to a line input with 'OK' (or a question mark '?' if there was an error). Later in the manual, the practice of underlining user input and indicating C64-FORTH's 'OK' response is discarded.

Numerical values in examples are sometimes given in DECIMAL, sometimes in HEX. We feel it is a great help to become proficient in the HEX number system as well as with decimal numbers. The relationship or necessity of using a certain value will often be far more obvious when displayed in HEX. The base of the values given in some examples is not stated when it is obvious what it must be.

LOADING AND RUNNING C64-FORTH

Make sure both the 64 and the disk drive are on before inserting the C64-FORTH disk into the disk drive.

Load the program by typing:

LOAD"C64FORTH",8 <return>

and when finished, type:

RUN <return>

A copyright and serial number message should have been printed and the cursor should be flashing. If not, remove the disk, turn power off to both the 64 and the disk drive for a few seconds, and repeat.

MAKING A BACKUP COPY OF C64-FORTH

Upon entering C64-FORTH for the first time, you should immediately make a backup copy of the C64-FORTH disk. With the original C64-FORTH disk still in the drive, type:

2 LOAD <return> (note: the space between '2' and 'LOAD' is important!)

A backup routine will be loaded from disk. Follow all instructions and when done you will have an exact copy of the C64-FORTH disk. You should put the original C64-FORTH disk away, and should use ONLY the backup from now on. Sections in this manual cover how to create an application program disk, how to backup source code screens that you create, and initialize a new screen disk. But until you become more familiar with C64-FORTH, the backup disk you just made is all you need.

CONTENTS

The Fundamentals of FORTH.....10

Entering Commands in the FORTH System.....14

Basic Stack Operations.....15
- manipulating stack items 16
- arithmetic operations 18
- working in different number bases 19
- constants and variables 20
- fetching and storing 21
- fetching and storing bytes 22
- double precision numbers 23

Flags and Comparisons.....25

Looping and Branching.....27
- do loops 27
- if..else..then 30
- begin..until 31
- begin..again 31
- begin..while..repeat 31
- nesting of looping structures 32

Arrays.....33
- simple arrays 33
- create..does> 34

Creating New Definitions.....38
- colon definitions 38

Text and Numeric Output.....41
- single character output 41
- text output 41
- numeric formatting 43

Text and Numeric Input.....44
- single key input 44
- inputting a line of text 45
- numeric input 47

Handling Strings in FORTH.....49
how strings are stored in memory 49
description of string commands 50 thru 59

File I/O.....60
description of file i/o commands 60 thru 66
sample programs using string and file i/o commands 66,67

Details of the C64-FORTH System.....68
SAVESYSTEM and DC 68
DIR 69
listing multiple screens on the VIC-1525 printer 69
BUFFERS, MEMTOP, and using available RAM 70
BORDER, BKGRND, and CHRCLR variables 72
restore key function 72
function of DV1, DV2, DV3, and DV4 73
function of SYSDEV# 75
TAPEFLG 75
validating the screens disk, and disk peculiarities 76
error messages 76
using IEEE interface cartridges 77
the structure of a definition in memory 78
debugging FORTH routines using the TRACE feature 80
memory map 82
SAVETURNKEY 83

The EDITOR.....A-1
editor command summary A-6

The ASSEMBLER.....B-1
assembler glossary B-22

FORTH Graphics on the Commodore 64.....C-1
graphics glossary C-3

Floating Point (Real) Numbers.....D-1
floating-point glossary D-4

C64-FORTH GLOSSARY.....E-1

THE FUNDAMENTALS OF FORTH

The FORTH language architecture is built around the DATA STACK. While FORTH allows memory locations to be assigned for use as variables, constants, and work areas, the data stack is used to hold and transfer most values and parameters (DATA ITEMS) for use by the various sections of a program. The data stack allows very quick and convenient access to data items. Since most data items needed by or produced within intermediate portions of a program are temporary in nature, the data stack proves very handy.

The way the data stack works is identical to the way Reverse-Polish-Notation (RPN) type of calculators on the market work. Many analogies have been created to describe the FORTH data stack in use, the following is perhaps one of the closest.

We all have a desk or table somewhere with a pile of stuff on it. When bills come in, we put them on 'top of the stack'. When we get paid, it's time to pay the bills (after all, we are responsible). We take the first bill off the top of the stack and take care of it. We continue until all bills have been removed from the stack and taken care of. A new bill has come in, and there is only one copy of it. We make a photocopy of it and throw that on the pile too so we have a copy for ourselves when we go to pay it.

This basic throw-on-take-off principle is referred to in FORTH as push-on-pop-off. We will refer to data put on the stack as DATA ITEMS. A data item, when added to the stack, is PUSHED ON. A data item, when removed from the stack, is POPPED OFF. In a FORTH program you will usually push appropriate values or data items on the stack, CALL (invoke) a lower-level procedure which will remove or modify values on the stack, and control will return to the higher-level procedure with results left on the stack.

The stack is the mechanism by which portions of a FORTH program communicate with other portions of the program. The FORTH language itself is comprised of a number of commands that operate on memory locations such as variables, constants, and buffers, or on data items on the stack. In other computer languages, these are referred to as instructions. In FORTH, the commands are usually referred to as WORD DEFINITIONS, or more simply WORDS. The number of words making up the FORTH language is variable. The FORTH standards define a minimum subset, and new words can easily be added by the user to make a very versatile and customized higher-level language. The concept of the FORTH word fits well with the concepts of the FORTH vocabularies and dictionary, which will be explained in a minute.

The FORTH program itself is loaded into the COMMODORE-64 just like a BASIC program. FORTH is different however in that a portion of itself is written in FORTH code. FORTH actually is comprised of two types of code. The first is direct machine code routines, just like what BASIC itself is written in. The second type is WORD DEFINITIONS. A WORD definition is comprised of a number of POINTERS. The pointers point to locations that contain addresseses of machine code routines. When a definition EXECUTES, the machine code at the address found at the destination of the first pointer is executed, then the code at the address found at the destination of the second pointer is executed, etc., giving the appearance that the WORD definition is executing.

A word definition has two parts. The HEADER is made up of the definition's NAME, as well as other information. The BODY of the definition contains pointers to addresses of routines that will execute specific functions. Definitions are linked together sequentially in the DICTIONARY. The dictionary may be comprised of several VOCABULARIES. FORTH is the name of the main vocabulary. To see the names of all the words in the FORTH vocabulary, type:

```
FORTH  VLIST   <return>
```

The listing on the screen may be slowed down by pressing the CTRL key on the C64 keyboard, or may be terminated by pressing the STOP key.

In C64-FORTH as shipped, there is only one other vocabulary, the EDITOR vocabulary. To see the words in the EDITOR vocabulary, type:

```
EDITOR  VLIST   <return>
```

Notice that some new names appear, followed by the words in the FORTH vocabulary. All vocabularies link back to FORTH. To return to just the main FORTH vocabulary, type:

```
FORTH     <return>
```

Why are new vocabularies created? You can create new definitions with the same names as words in the FORTH vocabulary. If you create those definitions in the FORTH vocabulary, the original definitions are no longer accessible. If you create them in a new vocabulary, they are. Also, as you enter commands (the names of words), the FORTH system searches through the current vocabulary, then the FORTH vocabulary itself, for a match. If a match is found, the definition is executed. Putting the definitions of seldomly used words into other vocabularies will speed up the searching when just the FORTH vocabulary is selected.

Even though the use of vocabularies suggests a 'tree' structure for the dictionary, it actually is a linear structure. When a new definition is added to the dictionary, it is put onto the end (top) of the dictionary, and information is put into its header that links (logically connects) it to the previously defined word IN THE PRESENT VOCABULARY.

Words are provided to do simple operations on data items on the stack and on the contents of memory locations. You can create and use variables, constants, data arrays, and buffers much the same as in other languages. There is a class of words available called DEFINING WORDS. These words are for creating new WORD TYPES or DATA TYPES. Unlike some other languages, you may create new types of instructions, and very complex custom data storage and manipulation commands. FORTH can be customized to be a very simple, easy to use language, or very complex with extensive high-level capabilities.

FORTH, unlike PASCAL or FORTRAN, is not a strongly 'typed' language. This means that in FORTH, you may freely manipulate values on the stack or in memory considering them as one type of data (e.g. 16-bit integers), then later you may use the data considering it to be another type (e.g. 2-byte pairs of ASCII characters).

In FORTH, unlike most other high-level languages, you have complete access to, and direct control of, the entire computer down to the machine level. This does provide extreme versatility and speed, however there is one major disadvantage - there is no protection of the FORTH system itself. It is quite easy, if one is careless, to overwrite parts of the FORTH system causing a system crash and loss of all recent work. The point here is that until you become an experienced FORTH user, save current work frequently. We don't want to discourage experimentation; we just suggest that you isolate it from useful work.

In other languages, you create new programs by writing source code as one long continuous file. A compiler, such as PASCAL, takes the source code file, reads it in, and converts it to executable machine code (called COMPILING). This machine code is saved as a file, and when the user wants the program to run, he loads the code file and executes it. With interpreters, such as the BASIC in the C64, the source code file is also the executable code in a way. The interpreter scans the program source code file and INTERPRETS (executes machine code routines that performs the function identified by the name of the instruction) each instruction name found. The program source code is one continous file.

FORTH functions in both a compiler and an interpreter mode. Normally, it is in the interpreter mode. When it responds to a line of commands after the RETURN key is pressed, FORTH acts very much like BASIC in the direct mode. FORTH tries to match each word name encountered. The code corresponding to the name is executed as each name is found in the INPUT STREAM. 'Input stream' is another name for the line of commands and/or numeric values that was input in response to the flashing cursor. Only when new words are being created by defining COLON DEFINITIONS (colon defs are explained in the section ADDING NEW DEFINITIONS), does FORTH enter the compiler mode. In this mode, word names are not interpreted as they are being read in from the input stream, but are compiled into memory. This means a new word is being created and added to the dictionary that may be executed later, when back in the interpreter mode, just by typing its name.

The source code for a new program is not written and saved as one continous file however. The concept of SCREENS thus arises. A FORTH screen (not to be confused with the screen of the TV or display that the 64 is hooked up to) is a block of 1024 bytes of data that is transferred to and from the disk or cassette as one unit. When a FORTH screen is LOADED, it is read in from disk, and the FORTH interpreter uses it as the input stream. In effect, it is the same as if you had typed in the 1024 characters in response to the flashing cursor.

By using the editor supplied with C64-FORTH, the user may place into screens the source code to define new word definitions. Since in FORTH the concept of writing a program is to keep definitions short and build up higher-level definitions from lower-level definitions, the 1024 byte limit on a screen is no problem. One screen may itself load another screen. Therefore, one entire program, comprised of source code on many screens, may be loaded with just one command.

Screens may also be used for purposes other than source code definitions. Since the screen, when loaded, is fed into the input stream, any words that aren't part of new colon definitions being created are executed immediately. Thus, a screen may act as an indirect command file. Direct reading and writing of screen blocks to and from memory is possible, allowing use of screens for storing data arrays, tables, machine code subroutines and overlays, etc.

C64-FORTH is shipped with a number of screens on the disk or cassette. Screens 4 and 5 contain ASCII error and status messages, screens 6 through 17 contain the source code for the assembler, and other screens may contain code or data.

ENTERING COMMANDS IN THE FORTH SYSTEM

Before continuing, how to enter commands in the FORTH system should be covered. Whereas in other languages, you have INSTRUCTIONS, OPERANDS (such as an immediate numeric value, name of a variable, etc.), and OPERATORS (such as + and =). In FORTH, you just have WORDS and NUMERIC VALUES. A WORD is an entry in the dictionary. When the name of a word is specified, a piece of code somewhere gets executed. Therefore, to effect a command, you just enter into the input stream the name of a word in the dictionary. You may enter the names of several words at a time in a command (input) line, or just one at a time. Every function available in FORTH has an entry in the dictionary and can be invoked just by entering the name of it.

When you press the RETURN key, all characters on the current line are passed to the FORTH as the input stream. Every word name or sequence of characters representing a number MUST be seperated by AT LEAST ONE SPACE. Since an operator such as multiply (*) in other languages is defined as a word in FORTH, there must be at least one space between numbers and things like + , * , etc. The only way FORTH knows when the name of one word stops. and the name of another word begins, or when a numeric representation begins and ends, is by detection of a space character.

You may be wondering what is the range of characters you may use within the name of a word. The answer is ANY SCREEN-PRINTABLE character may be used within the name of a word. To keep compatability with other FORTH systems however, you should not use any special Commodore graphics characters, only standard ASCII characters when defininging new dictionary entries. The name of any FORTH word may be up to 31 characters in length.

How do you enter comments between FORTH commands (word names) ? Merely by placing the comment within parentheses. But remember, you must place at least one space between the left-parenthesis and the start of the comment, since "(" is a word, a dictionary entry, that when executed, scans off the input stream up to the first right-parenthesis found (No, nesting of comments is not allowed in FORTH).

Why is FORTH called a 'postfix notation' language ? What this gibberish means is that any parameters, qualifiers, numerical values, etc. that a word will need MUST be on the stack BEFORE the word is executed. In BASIC, you specify the instruction first, then the parameters, such a numerical values, the string to print, variable names, etc. Just remember that in FORTH you must specify everything first, then you specify the name of the word you want to execute.

BASIC STACK OPERATIONS

In this manual, the term DATA ITEM is used generically to refer to an item on the stack. A data item in most cases takes up 16-bits (2-bytes) of stack space. Getting a number on the stack is quite simple - just specify it. A string of characters found in the input stream that does not match any entry in the dictionary is converted to a number (if possible) and pushed onto the data stack. The range allowed for number input is -32768 to +65535 (ways of getting around the limits will be discussed later). A number in this range is called a SINGLE PRECISION INTEGER.

Words that operate on items on the stack consider single precision integers to be one of two types:

1) a SINGED SINGLE PRECISION INTEGER. This is a whole number in the range of -32768 to 32767.

2) an UNSIGNED SINGLE PRECISION INTEGER. This is a whole number in the range of 0 to 65535. An example where an unsigned integer will be specified is the address required for fetching data from or storing data to memory or I/O devices.

Note that the unsigned value 65535 is equal to the signed value of -1, 65534 is equal to -2, ... and unsigned 32768 is equal to signed -32768. It is up to the routine that uses or modifies a stack or memory value, as to how the value is interpreted.

To put a number(s) onto the stack, just type in the value(s) in response to the flashing cursor, followed by <return>. An 'OK' is printed. The 'OK' is FORTH's way of showing that the line just input was fully accepted. To print out the top value on the stack, use the "dot" command (the ascii decimal point, or period). Dot removes the top value and prints it as a SIGNED single precision integer.

142 OK

. 142 OK

-7 678 10426 -32000 OK

. -32000 OK

. . 10426 678 OK

. -7 OK

32769 . -32767 OK

Note that the most recently added value to the stack is printed first. At this point, the stack should be empty. To verify, just enter the dot command again.

```
. 0  EMPTY STACK
```

If the empty stack message didn't occur, you somehow had extra values on the stack. Keep executing the dot command until the message occurs. The value printed just before the empty stack message is always meaningless, since the check for the stack being empty is always made upon return back into the input prompt routine.

The word used to print the top stack value as an UNSIGNED integer is U. (pronounced "u dot").

```
-1 U.  65535  OK
```

- MANIPULATING STACK ITEMS

There are several FORTH words available for duplicating, removing, or changing the position of stack items. To assign an identity to different stack items for the purpose of showing effects on stack items by FORTH words, a certain notation is used. This is described more fully on page 1 of appendix E, the start of the glossary. Basically, n1 means number #1, n2 means number #2, etc. If n2 is to the right of n1, then n2 is higher on the stack (closer to the top) than n1. The three dashes ("---") indicate the execution point of the word. Values to the left of the dashes are what's required on the stack before the word is executed, values to the right are the results after execution of the word.

Mention should be made here as to the function of the GLOSSARY at the back of this manual. The glossary works like a normal dictionary. The names of all the words in the base FORTH vocabulary are listed, with descriptions of how the words function, and their stack requirements and effects. Once you get up to speed in FORTH, you will probably be using only the glossary part of this manual. There are seperate glossaries for words available for graphics use, strings, floating-point, and under the ASSEMBLER. These have been placed in seperate sections of the manual since they are not necessary in normal FORTH programming.

The stack manipulation commands are:

DUP n1 --- n1 n1
duplicates the top stack item

SWAP n1 n2 --- n2 n1
exchanges the top two stack items

DROP n1 ---
removes and discards the top stack item

OVER n1 n2 --- n1 n2 n1
copies the second item and pushes it on the stack

ROT n1 n2 n3 --- n2 n3 n1
rotate the top three stack items

n PICK ... n4 n3 n2 n1 n --- ... n4 n3 n2 n1 n?
copy the n-th stack item (not counting n itself) and push it onto the stack. e.g. 1 PICK is the same as DUP, 2 PICK is the same as OVER.

n ROLL ... n4 n3 n2 n1 n --- ... n? n? n? n?
rotate the top n stack items (not counting n.itself) e.g. 3 ROLL is the same as ROT, and 4 ROLL has the effect: n4 n3 n2 n1 4 --- n3 n2 n1 n4

Examples:

37 DUP . . 37 37 OK

123 19 SWAP . . 123 19 OK

42 12 DROP . . 42 0 EMPTY STACK

78 6 OVER . . . 78 6 78 OK

10 20 30 ROT . . . 10 30 20 OK

10 20 30 40 4 PICK OK

. . . . . 10 40 30 20 10 OK

10 20 30 40 4 ROLL OK

. . . . 10 40 30 20 OK

- ARITHMETIC OPERATIONS

It is very simple to perform arithmetic operations on values on the stack. All values necessary must be on the stack before the operation is performed:

```
3 17 + .  20  OK

20 32 - .  -12  OK

7 8 * .  56  OK

204 17 / .  12  OK
```

FORTH is a 'free-format' language. You may specify one operation per line or many operations per line. You may also control when operations are done, as long as the required number of values are on the stack:

```
3 7 2 8 + + + .  20  OK

3 7 + 2 + 8 + .  20  OK

3 7 + 2  OK

8 + + .  20  OK
```

Operations are performed in the order that they are specified:

```
3 7 8 + * .  45  OK              ( equiv. to (7+8)*3 )

3 7 8 * + .  59  OK              ( equiv. to (7*8)+3 )
```

There are several other words in FORTH for arithmetic operations. Refer to the glossary for descriptions on:

```
*/    */MOD    /MOD    MOD
1+    1-    2+    2-    2*    2/
ABS    MAX    MIN    NEGATE
```

- WORKING IN DIFFERENT NUMBER BASES

Numbers in base 10 (the decimal system) are not at home in computers. The relationship between addresses, bit patterns, displays of memory contents, etc. become more apparent when working in other number bases. While decimal numbers are sufficient in BASIC, for languages such as FORTH or assembly that allow you more absolute control over the computer hardware, HEXadecimal (base 16) is more meaningful.

BASE is a FORTH variable that keeps track of the current number base. All input and output number conversions (characters into binary values and visa versa) are done according to the current base.

DECIMAL sets BASE to 10. All numbers input are expected to be decimal numbers, and all numbers printed out are decimal.

```
DECIMAL 8 9 + .  17  OK
```

HEX sets BASE to 16:

```
HEX 8 9 + .  11  OK

1A C + .  26  OK
```

The current BASE selected stays in effect until changed. If you were brought-up in mini-computer land, you are probably used to OCTAL numbers. You may work with octal numbers by setting BASE to 8:

```
8 BASE ! 7 6 + .  15  OK
```

Other bases (such as BINARY) may be selected in a similar way (2 BASE !).

- CONSTANTS AND VARIABLES

In BASIC, a constant is a variable whose value does not change throughout the course of the program. In FORTH, CONSTANTS and VARIABLES are uniquely different data types.

Constants and variables are dictionary entries, just like word definitions. When the name of a constant is executed, the value of that constant is pushed onto the stack. The numbers 0, 1, 2, and 3, because they are used so frequently, are actually defined as constants. Since the dictionary is searched before a numeric conversion is attempted, this speeds things up.

There are other constants in FORTH, such as C/L. C/L, when executed, pushes onto the stack the number of characters per display line:

```
DECIMAL  C/L  .  40  OK
```

The word CONSTANT itself is a defining word. It is used to create new constants:

<initial value> CONSTANT <constant name>

Example:

```
7  CONSTANT  DAYS/WK  OK
```

creates a new dictionary entry, called DAYS/WK , and assigned the value of 7 to it. Whenever DAYS/WK is executed, the value 7 will be pushed onto the stack:

```
DAYS/WK  .  7  OK
```

VARIABLES are used extensively in FORTH. As in BASIC, variables may be initialized and changed easily. When the name of a variable is executed, the ADDRESS (location) of the contents of the variable is pushed onto the stack.

In the dictionary, a constant or variable is made up of a header (containing the name and linkage and execution-code pointers), and a body consisting of two bytes. The 16-bit value of the constant or variable is stored in these two bytes. The only difference is that a constant will fetch the contents of the body and put it on the stack, and a variable will put the address of the body (location of the value) on the stack.

The word VARIABLE itself is a defining word. It is used to create new variables:

VARIABLE <variable name>

Example:

VARIABLE #OFERRORS OK

When #OFERRORS is executed, it leaves the address of its current value. When a variable is created, its initial value is random. To initialize it, a value must be stored in it as described in the following section.

In FORTH, there are a number of USER VARIABLES (e.g. BASE). The contents of the variable are not stored immediately following the header in memory, but in a table called the USER AREA located at the top of memory used by the FORTH system. Although functionally identical to variables, executing the name of a user variable leaves the address within the user area of the contents of the variable.

- FETCHING AND STORING

Two words, @ ("fetch") and ! ("store") are used to read from, or write to variables or memory locations. @ 'fetches' (reads the 16-bit contents of) the memory location pointed to by the address on the stack. The address on the stack is replaced by the contents read from that location:

641 @ . 2048 OK

The equivalent in BASIC is:

PRINT(PEEK(642)*256+PEEK(641))

Since executing the name of a variable leaves an address on the stack, then @ can be used to read the contents of a variable.

From a previous example:

#0FERRORS @ . 0 OK

(the value printed may not be 0 since we did not initialize the variable #0FERRORS yet.)

BASIC:

200 PRINT NE

User variables may be read the same way:

BASE @ . 10 OK

! (store) is used to write a value to a 16-bit memory location. ! requires two values on the stack. The second stack value must be the value to store, and the top stack value must be the address to store it at. Both values are removed from the satck after the store is done.

513 1142 ! 0 55414 ! OK

The letters "AB" should have appeared in black in the upper right corner of the display screen. Since executing the name of a variable leaves an address on the stack, then ! can be used to change the contents of a variable:

10 BASE ! OK

- FETCHING AND STORING BYTES

@ and ! are used for 16-bit values. Since the 6502/10 is an 8-bit machine, words are provided for reading from and writing to byte locations. C@ is used for 8-bit fetching, and C! is used for 8-bit storing.

(what is the current border color ?)

53280 C@ . 14 OK

(change the current border color)

4 53280 C! OK

(restore the border color to start-up value)
(Note: the variable BORDER contains the color value that C64-FORTH sets the border to upon start-up)

BORDER C@ 53280 C! OK

- DOUBLE PRECISION NUMBERS

Since each position on the stack can hold a data-item consisting of 16-bits, the number range is limited to -32768 to 65535. This is rather limited for some types of applications, so a class of words is provided in FORTH to extend the range of possible stack values. Called DOUBLE PRECISION INTEGERS, the effective range may be -2,147,483,648 to 2,147,483,647. Only SIGNED double precision integers are supported in this version of C64-FORTH. Double numbers take up 32-bits (4-bytes) of stack space.

A double number is input by placing a decimal point somewhere within the number. No matter where the decimal point is placed, the same value ends up on the stack, but the location where the decimal point was found is placed in the user variable DPL.

The word for removing the double number on top of the stack and printing it as a SIGNED double integer is D. ("d dot"):

```
1234.567 D.  1234567  OK

DPL @ .  3  OK

12.34567 D.  1234567  OK

DPL @ .  5  OK

123456.7 D.  1234567  OK

DPL @ .  1  OK
```

A double number on the stack may be treated as two single integers. The high order part of the number (the high 16-bits) is the top stack item, the low order part is the second stack item:

```
HEX 12345678.  OK

.  1234  OK

.  5678  OK
```

To convert a positive single integer on the stack to a double integer, just push a zero. To convert a negative single integer on the stack to a negative double integer, just push 65535. The word S->D (for 'single-to-double') automatically converts a single precision integer into a double integer, accounting for the sign of the integer. To convert a double integer into a signed single integer, just execute DROP to remove the top 16-bits of the double number.

Double number equivalents to most of the single integer stack manipulation commands exist. They are the same commands preceeded by a '2'.

```
2!   2@
2DUP   2DROP   2SWAP   2OVER   2ROT
```

2CONSTANT and 2VARIABLE work similar to their single number counterparts:

```
DECIMAL  8732109.  2CONSTANT  BIGNUM  OK

BIGNUM  D.  8732109  OK

2VARIABLE  BNUM  OK

99998888.  BNUM  2!  OK

BNUM  2@  D.  99998888  OK
```

Double precision arithmetic operations usually begin with a 'D'.

```
D+   D-  DABS   DMAX   DMIN   DNEGATE
```

There are several words used that accept or produce both single and double precision numbers. Refer to the glossary for their functions:

```
D+-   M*   M/   M/MOD   S->D   U*   U/
```

FLAGS AND COMPARISONS

- FLAGS

FORTH uses flags extensively. Actually, a flag placed on the stack is just a numeric value. If the value placed on the stack is a zero (0), it is said to be a FALSE flag. If a value placed on the stack is non-zero, it is said to be a TRUE flag. A true flag is usually a value equal to 1, but not always. Many FORTH routines return a condition, either true or false, to indicate whether something did or did not happen, something was or was not within a specified range, etc. Program branching and looping directives, described later, may test the value of the flag, and take action depending on its state.

- COMPARISONS

Several FORTH words are provided for doing comparisons between numeric values on the stack. A flag is returned by the comparison routine. In the case of 0= , a true flag is left on the stack if the top stack value when the routine 0= was called was equal to zero (0). 0= removed the top stack value in the process of testing it. If the value was not equal to zero, the flag returned is false, or equal to 0. 0= can thus be seen to be useful for inverting the TRUTH of a flag, i.e. using 0= on a flag on top of the stack will return a FALSE value of the flag was TRUE, and will return a TRUE value if the flag was FALSE. (NOT is another FORTH word that is identical to 0=)

There are two other comparators that test just one stack value. They are 0< which tests if the top stack value is less than 0, and 0> which tests if the top stack value is greater than zero. Both of these assume the top stack item is a signed single integer.

```
1 0= .  0  OK
0 0= .  1  OK
-2 0< .  1  OK
28 0> .  1  OK
65530 0> .  0  OK
```

There are four comparators that operate on the top two stack values: < = > and U< . < and > assume the two values are signed single integers, U< assumes they are unsigned single integers, and = just plain doesn't care.

The relationship tested between comparators that require two stack items is:

<second stack value> COMPARISON <top stack value>

Thus, 3 18 > will return a FALSE flag since 3 is not greater than 18.

There are five comparators for testing the relationship of signed and unsigned double precision integers:

D0= D< D= D> DU<

Refer to the glossary for descriptions.

LOOPING AND BRANCHING

- DO LOOPS

A DO loop is similar to the FOR..NEXT loop in BASIC. A loop is set up with a starting value, an ending value, and an increment. The value that changes with each pass through the loop is called the INDEX of the loop, and the value that serves to terminate (cause the exiting of) the loop is called the LIMIT.

A DO loop is constructed as follows:

```
...  <limit+1>  <starting value>  DO  .. loop code ..  LOOP
```

A value that is equal to the desired loop limit +1 is pushed onto the stack, the value representing the starting value of the index is pushed onto the stack, then the loop is entered. It is important to understand that the words DO and LOOP can only be used inside of a colon definition, because at compile-time they create the structure that at execute-time will provide a looping effect!

The limit must be specified as 1 greater than the value desired to terminate the loop since the index is incremented BEFORE it is compared against the limit.

As an example, lets print the first 5 characters of the alphabet. Since a DO loop can only be implemented within a colon definition, we must define a word with the loop and then execute the word:

```
: ALPHA5  70  65  DO  I  EMIT  LOOP  ;

ALPHA5  ABCDE  OK
```

The FORTH word I fetches the current index of the loop and pushes it onto the stack. The equivalent of the above routine in BASIC is:

```
10 FORI=65TO69
20 PRINTCHR$(I);
30 NEXTI
```

DO loops may be nested:

```
DO . . DO . . DO . . . . LOOP . . LOOP . . LOOP
```

To obtain the current index of the next outer loop the FORTH word J is used, and to obtain the index of the third outer loop K is used:

```
: LOOPTEST  3  0  DO
                2  0  DO
                  CR  ." I="  I  .  ." J="  J  .
                LOOP
            LOOP  ;
```

```
LOOPTEST
I=0 J=0
I=1 J=0
I=0 J=1
I=1 J=1
I=0 J=2
I=1 J=2  OK
```

In BASIC, the above routine would be:

```
10 FORJ=0TO2
20 FORI=0TO1
30 PRINT"I=";I;"J=";J
40 NEXTI
50 NEXTJ
```

LOOP sets up the structure of the definition being created such that at execute-time a value of +1 is added to the current index, then the index is compared with the limit. A different increment, including negative values, may be specified by using +LOOP instead of LOOP. +LOOP takes the top stack value and adds that to the loop index. The loop index is then compared against the limit, and if it exceeds the limit, the loop is exited:

```
:  MINUSSTEP  -1  100  DO  CR  I  .  -10  +LOOP  ;
```

is equivalent to:

```
10 FOR I= 100 TO 0 STEP-10
20 PRINTI
30 NEXTI
```

NOTE: both LOOP and +LOOP work according to SIGNED arithmetic. This means that the loop will be terminated either by the current index exceeding the limit, OR arithmetic overflow occuring when COMPARING THE CURRENT INDEX TO THE LIMIT. For instance, if the limit of the loop was specified as a negative number, and the current index is being incremented in the positive direction, the loop will terminate after only one iteration. Similarly, if the limit was specified as positive, and the current index is increasing in the negative direction, the loop will terminate after only one iteration.

A DO loop may be prematurely terminated by executing LEAVE. LEAVE sets the current index equal to the loop limit. However, the DO loop IS NOT EXITED UNTIL THE LOOP POINT IS ENCOUNTERED. Since desire to terminate a DO loop prematurely is probably based upon some event happening, LEAVE can be executed within the IF..ELSE..THEN structure that detected the condition:

```
DO .. <flag> IF LEAVE ELSE ..loop code.. THEN LOOP
```

Since LOOP and +LOOP work according to SIGNED arithmetic, this means that absolute memory addresses cannot be used for the index and limit of a loop if the address range crosses over the boundary at 32768 (8000 hex). There is a word that has been PROPOSED as a future FORTH standard word called /LOOP . /LOOP works like +LOOP but according to UNSIGNED arithmetic, thus absolute memory addresses may be used. The current FORTH PROPOSAL states that /LOOP is to work only with the index being incremented in the POSITIVE direction. /LOOP as provided in this implementation of C64-FORTH works in BOTH directions to allow you to increment the index in both positive AND negative directions.

Examples of usage (all numbers are in HEX):

```
... 8020 7FF0 DO I C@ FF XOR I C! 3 /LOOP ...
```

will complement every third byte in the range of 7FF0 to 801F

```
... 77FF 8FFF DO 0 I C! -1 /LOOP ...
```

will clear all bytes in the range 8FFF down to 7800 starting with the highest byte first.

- IF...ELSE...THEN

Since FORTH does not allow line numbers or labels, structures have been provided for program branching. IF..ELSE..THEN is used as follows:

```
<flag>  IF  true part  ELSE  false part  THEN
```

The IF..ELSE..THEN structure can only be used within a colon definition. At execute-time, a flag is expected on the stack upon entering the structure. The code that was compiled into the definition by IF tests the flag. If it represents a TRUE condition, then the code between IF and ELSE (the true part) is executed. If the flag is FALSE, then the code between ELSE and THEN (the false part) is executed. Either the true part OR the false part is executed, not both. Whichever part was entered, execution continues after THEN.

```
: TESTFLAG  ( print whether flag on stack is true or false )
    IF  ." FLAG IS TRUE"
    ELSE  ." FLAG IS FALSE"  THEN  ;

0 TESTFLAG  FLAG IS FALSE  OK

1 TESTFLAG  FLAG IS TRUE  OK
```

BASIC equiv:

```
10 IF C=0 THEN 40
20 PRINT"C IS TRUE"
30 GOTO 50
40 PRINT"C IS FALSE"
50 REM ** REST OF PROGRAM **
```

IF..THEN is a special case allowing execution of a section of code only if a flag on the stack tests TRUE. If FALSE, execution just skips over the IF..THEN part.

```
: TEST1  IF  ." FLAG IS TRUE "  THEN  ." THIS IS COMMOM CODE"  ;

0 TEST1  THIS IS COMMON CODE  OK

1 TEST1  FLAG IS TRUE THIS IS COMMON CODE  OK
```

BASIC equiv:

```
10 IF C=0 THEN 30
20 PRINT"C IS TRUE"
30 PRINT"THIS IS COMMON CODE"
```

- BEGIN..UNTIL

BEGIN..UNTIL is handy for looping until a certain condition exists. The condition (flag) is tested just before the UNTIL part:

```
BEGIN  loop code  <flag>  UNTIL
```

The loop code is always executed at least once. Some comparison or operation is done at the end of the loop code that leaves a flag on the stack for the UNTIL part to check. If the flag tests FALSE, execution loops back to just after the BEGIN part. If the flag tests TRUE, execution falls through to after the UNTIL part.

```
( wait until the 'Y' key is pressed )
:  BTEST  BEGIN                    ( start of loop )
            GET                    ( read keyboard )
            89  =  UNTIL  ;       ( Y key = value of 89 )
```

BASIC equiv:

```
10 GETA$
20 IF A$ <> "Y" THEN 10
```

- BEGIN..AGAIN

BEGIN..AGAIN creates what amounts to an infinite loop:

```
BEGIN  ..loop code..  AGAIN
```

The loop code is executed forever. While this may not immediately seem useful, certain process control or turnkey system applications will need this structure.

- BEGIN..WHILE..REPEAT

BEGIN..WHILE..REPEAT forms a structure that is an infinite loop, but can be exited upon a certain condition:

```
BEGIN  ..part1..  <flag>  WHILE  ..part2..  REPEAT
```

When the BEGIN..WHILE..REPEAT structure is entered, part1, if it exists, is executed. A flag placed on the stack by a comparison or operation is tested. If the flag is TRUE, the following code (part2) is executed. If the flag tests FALSE, the loop is exited and execution continues after the REPEAT. If part2 is executed, then control jumps back to just after the BEGIN.

Once again, any control structure (DO..LOOP, IF..THEN..ELSE, and BEGIN.. structures) can only be used within a colon definition! Any attempt to use them while in the interpretive mode will generate an error.

- NESTING OF LOOPING STRUCTURES

Nesting of looping and branching structures is allowed as long as EACH STRUCTURE IS FULLY CONTAINED WITHIN ANOTHER. For instance, an IF..THEN structure cannot be used to branch out of a DO loop or BEGIN..UNTIL structure:

```
DO  ..  IF  ..  LOOP  THEN                    NOT ALLOWED!
BEGIN  ..  IF  ..  UNTIL  THEN                NOT ALLOWED!
IF  ..  BEGIN  ..  THEN  .. UNTIL             NOT ALLOWED!
```

These are examples of allowable structures:

```
BEGIN  ..  IF  ..  ELSE  ..  THEN  ..  UNTIL
DO  ..  BEGIN  ..  WHILE  ..  REPEAT  ..  LOOP
IF  ..  DO  ..  BEGIN  ..  UNTIL  ..  LOOP  ..  THEN
```

ARRAYS

- SIMPLE ARRAYS

Looking at how FORTH works, we see three things that must be done to create and use arrays:

1) The amount of space necessary must be reserved in the dictionary so FORTH does not overwrite the contents of the array

2) A header and name must be given to the array so we may obtain the address that the data starts at

3) Initializing or changing array entries must be easy

CREATE is a FORTH word used to create a header, assign it the following name, and link it to a code routine that when executed, leaves the address of the first byte in the body. Used as follows:

CREATE <defname>

A new definition, called <defname>, is added onto the end of the dictionary, and is linked to the current vocabulary. There is no space allocated for the body of <defname>. When <defname> is later executed, the address of the first byte after its header is pushed onto the stack.

Space may be allocated with the ALLOT command. ALLOT takes a specified number, and adds it to the end-of-the-dictionary pointer. The contents of the space that is allocated is not initialized or altered in any way.

Instead of ALLOT, we may COMPILE specific values into memory. This process places an 8 or 16-bit value onto the end of the dictionary, and advances the end-of-the-dictionary pointer by 1 or 2. C, compiles a byte value into memory advancing the dictionary pointer by 1, and , (comma) compiles a 16-bit value into memory advancing the dictionary pointer by 2. With C, and , we now have a way to allocate and initialize memory at the same time.

Suppose we would like to create a table of squares for the first ten integers. In BASIC we would have:

```
10 DIMA(10)
20 FORX=0TO9
30 READN:A(X)=N
40 NEXTX
50 REM *** REST OF PROGRAM ***
1000 DATA 1,4,9,16,25,36,49,64,81,100
```

In FORTH:

```
CREATE SQUARES
1 C, 4 C, 9 C, 16 C, 25 C, 36 C, 49 C, 64 C, 81 C, 100 C,
```

A table of 10 bytes is now created. When SQUARES executes, the address of the first byte within the table is pushed onto the stack. Offsets may be added to this value to point to other values in the table:

```
SQUARES 2+ C@ .  9  OK

SQUARES 9 + C@ .  100  OK

150 SQUARES 8 + C!  OK

SQUARES 8 + C@ .  150  OK
```

Tables of 16-bit values may be created with , (comma) instead of C, . If the table need not be initialized, but will be filled by the program, just use ALLOT:

```
CREATE  SQUARES  10  ALLOT
```

- CREATE and DOES> structure

CONSTANT, VARIABLE, and ":" have been referred to as defining words, meaning that they are used to create new definitions in the dictionary. Most FORTH programmers can go the rest of their lives using just these three types for creating all new definitions. This explanation of the CREATE and DOES> structure can be skipped over by new FORTH users to avoid confusion.

DOES> is always used with CREATE within a colon definition. The CREATE - DOES> team is used to create new defining words (similiar to defining a new data type in some other higher level languages, such as PASCAL, but in FORTH it is really generating a new WORD TYPE). The greatest use for creating new word types with the CREATE - DOES> structure is for defining arrays.

CREATE and DOES> are always used together in the same definition. There are two seperate operations to consider about any defining word. The first is what happens when the defining word is used to create a new definition. The second is what happens when the newly-defined word executes.

CREATE - DOES> allows you to precisely define both circumstances. They are used as follows:

```
: <typename>  CREATE  ..part1..  DOES>  ..part2..  ;
```

A new defining word is created called <typename>. The new defining word can be used to create new word definitions or data types:

```
<typename>  <defname>
```

When <typename> is executed, a new definition is created with the name <defname>. When building the definition <defname>, the FORTH code between the CREATE and DOES> part of <typename> (..part1..), is executed. Typically, this code will allocate space for an array, and provide a way to initialize values within the array if desired.

When <defname> is later executed, the address of the first byte in the body of <defname> is pushed onto the data stack, THEN the FORTH code between the DOES> and ; part of <typename> (..part2..) is executed. Typically, this will provide the address of a specific entry within the table, provide the entry within the table, or somthing similar.

As an example, even though the defining word CONSTANT is already present within the FORTH vocabulary, we can redefine it:

```
: CONSTANT  CREATE  ,  DOES>  @  ;
```

When CONSTANT is used to define a constant:

```
12  CONSTANT  MON/YR
```

a new header is created in the dictionary with the name MON/YR. Then the part between CREATE and DOES> is executed. The comma takes the top value off the data stack, and compiles it into memory. The header called MON/YR now has a 16-bit value equal to 12 in the first two bytes after it.

When MON/YR is later executed, the address of the first byte of its body is pushed onto the stack. Then the part between DOES> and ; of CONSTANT is executed, which fetches the 16-bit contents of the location pointed by the top stack item, i.e. the address just pushed. Likewise, we can see that the defining word VARIABLE would be defined as:

```
: VARIABLE CREATE 0 , DOES> ;
```

Here the part between DOES> and ; is nonexistent, so nothing is done to the address left on the stack.

Now, to redefine our previous array SQUARES. We want to create a defining word that can be used to create arrays. We want it to be capable of creating and initializing an array of 10 entries in one case, 18 entries in another case, etc.. When we later refer to the name of a array created, we want it to return the address of a specific entry within the array.

Lets create our new defining word:

```
: ARRAY CREATE 0 DO C, LOOP DOES> + 1- ;
```

ARRAY is then used as follows:

```
100 81 64 49 36 25 16 9 4 1 10 ARRAY SQUARES
```

A new array called SQUARES is created. The first thing that happens after the header is formed, is the top stack value is taken as the limit of the DO loop. This should be the number of entries that will be in the array being created. Next, the DO loop is entered. For as many times as there are entries in the array, a value will be pulled off the data stack and compiled into the dictionary. This will effectively both allocate space for and initialize each entry. In this example, first the 1 is pulled off the stack and placed in the first entry, then a 4 in the second, etc.

When SQUARES is later executed in the following manner:

```
6 SQUARES
```

the address of the first entry of the table is pushed onto the stack. The part between DOES> and ; of ARRAYS is then executed, which adds the address to the previous value on the stack, in this case the 6. Since we are expecting the address of the 6th entry in the table, and entry 1 is in the address+0 position, a 1 is next subtracted from the effective address. The resulting address is the location of the 6th entry in the table. It may be accessed by:

```
6 SQUARES C@ . 36 OK
```

Other arrays may be created with the same defining word:

```
31 30 31 30 31 31 30 31 30 31 28 31 12 ARRAY DAYS/MONTH
```

The number of days in April is found as:

```
4 DAYS/MONTH C@ . 30 OK
```

To define ARRAY for creating arrays that don't need initialization:

```
: ARRAY CREATE ALLOT DOES> + 1- ;
```

and is used in the form:

```
10 ARRAY TEMPVALS
```

With techniques explained later, large arrays can be initialized from data stored in screens.

- COLON DEFINITIONS

The most common way to add new commands to the FORTH system is to create COLON DEFINITIONS. Most of the words in the base FORTH dictionary are defined as colon defs. A colon definition is essentially a list of pointers to previously-defined words. When FORTH appears to 'execute' a word that was defined as a colon def, actually it is executing the code (or colon def) identified by each pointer in the body of the definition.

A colon definition, as the name implies, starts off with a colon (':'). Next, the user supplies the name which is to be assigned to it. The names of previously-defined FORTH words make up the body of the definition, which is terminated with a semi-colon (';'):

```
:   <name>   . . . <body> . . .   ;
```

Example:

```
:   CLRSCREEN   147   EMIT   ;
```

A new word, named CLRSCREEN, has been added to the FORTH dictionary. To see this, type VLIST . CLRSCREEN is now the top word in the current vocabulary.

When 'CLRSCREEN' is typed in, the body of the definition will be scanned and executed. First, the value 147 is pushed onto the stack, then the routine EMIT is executed. EMIT takes the top value off the stack and prints it as a character (printing a value of decimal 147 happens to clear the screen).

Notice that when you were defining the new definition CLRSCREEN, the screen did not clear. Normally, if you type 147 EMIT in response to the flashing cursor, the screen clears. However, when a colon is encountered in the input stream, the FORTH system stops executing commands (interpreter mode) and starts COMPILING a definition (i.e. enters the compiler mode). A HEADER entry is created in the current vocabulary, the following <name> is assigned to it, linkage pointers are added, and all subsequent numeric values and previously-defined FORTH words up to the semi-colon are COMPILED into the definition. 'COMPILED into the definition' means a pointer is added onto the body of the current definition that either points to another definition to execute, or to a routine that at execute-time will push the specified numeric value onto the stack. Encountering the semi-colon turns off the compile mode and FORTH re-enters the interpreter mode. Any word names now found in the input stream will be executed immediately again.

Colon defs may be NESTED to any depth. This means once a definition has been created, it may be referenced in the body of a new definition, which may be referenced in the body of an even newer definition, etc. For example, having already defined CLRSCREEN, we may enter:

```
: 3RDLINE  CLRSCREEN  CR  CR  ;
```

When 3RDLINE is executed, the screen is cleared and the cursor is moved down to the third screen line. 3RDLINE, now defined, may be used in the definition of a new command, and so forth:

```
: SHOWNEST  3RDLINE  ." EXAMPLE OF NESTING"  ;
```

Most of the words in the FORTH glossary may be used inside of a colon definition. There are some words classified as IMMEDIATE. These words overide compile mode and execute immediately. Examples are the names of vocabularies (ASSEMBLER, EDITOR, etc.) and branching control structures (IF..ELSE..THEN, BEGIN..UNTIL, etc.). Words marked as IMMEDIATE are not compiled into the current definition, but perform a function necessary during the compile mode. Good examples of colon defs are the definitions of some of the FORTH words:

```
: DECIMAL  10  BASE  !  ;
: HEX  16  BASE  !  ;
: PAD  HERE  68  +  ;
: BLANKS  BL  FILL  ;
: DEFINITIONS  CONTEXT  @  CURRENT  !  ;
: UPDATE  PREV  @  @  32768  OR  PREV  @  !  ;
```

How many FORTH words can you use to make up the body of a new definition? As many as you want. It is customary in FORTH programming to keep definitions short, but it is not a rule. There is a limit when typing in an input line. 80 characters is the maximum the 64 allows you to input at one time. However, since FORTH is free-format, you may create a definition by inputting many lines. When entering a new definition longer than an 80-character input line, you'll notice that FORTH does not respond with the familiar 'OK' response when the RETURN key is hit. When you finally terminate the new definition by using the semi-colon, FORTH re-enters the interpreter mode and responds with 'OK'.

When you type in a colon definition as an input line, you loose the 'source code'. There is no easy way to examine a word definition to determine the sequence of words that was used to define it. Therefore, it is good practice to enter all definitions, constant and variable declarations, defining arrays, etc. onto source screens. The LOAD command reads in a source screen treating it as if it were typed in as one long input line. The advantage gained here is that it can be permanently stored onto disk or tape, aiding future work. The LOAD command is used as follows:

```
<scr#>  LOAD
```

The screen# <scr#> is read in from disk if not already in memory, and the FORTH input interpreter begins at the start of the screen considering it as a 1024 byte long input line. Normally, when the 1024 bytes are all read in, FORTH prints 'OK' and returns to expecting input from the keyboard buffer. If the word '-->' is found in the screen being LOADed, then the FORTH interpreter stops using the current screen, looks for the next screen in memory or on disk, then feeds that screen into the input stream. If an error is encountered anywhere, LOADing of the current screen immediately stops and an error message is printed. At this point, and before entering ANY other command, if you type WHERE <return>, FORTH will display the current line within the current screen the error occurred at, and with an up-arrow will point to just after the word/character string the error occured in.

Say you had a line in screen #20 that read:

```
  :  TEST  CR  XXX  ;
```

and you do not have a word already defined in the dictionary called XXX. When you go to load the screen you will get an error, and when you type WHERE you'll see the following:

```
20  LOAD  20  XXX  ?  CAN'T FIND
WHERE
  :  TEST  CR  XXX  ;
                ↑
```

Right after entering the LOAD command, FORTH echoes the screen number it is LOADing. When XXX was encountered, no entry was found in the dictionary, so it is printed out with the ? CAN'T FIND error message. When WHERE was typed in, FORTH printed the current line that contained the error, and on the following line printed an up-arrow just after the word that was not recognized.

Many of the screens on the C64-FORTH disk contain source code written in FORTH. Analysis of the routines plus experimentation is the best way to learn how to define new definitions. Create some of your own source screens using the editor (the editor is described in Appendix A) and try loading them.

TEXT AND NUMERIC OUTPUT

- SINGLE CHARACTER OUTPUT

A value on the stack may be printed as an ASCII character by using the word EMIT. The value to be printed should be in the range 0 to 255.

```
65 EMIT 67 EMIT AC OK
```

BASIC equiv:

```
PRINTCHR$(65);CHR$(67);
```

- TEXT OUTPUT

There is an easy way to print out ASCII messages. ." (dot-quote) is a FORTH word used to print out a string of characters. It is used as follows:

```
." <text string>"
```

There must be one space between ." and the start of the message. The text message is terminated by a quote character. ." prints out the character string starting one space after it up to the terminating quote character.

Example:

```
." TEXT STRING OUTPUT TEST" TEXT STRING OUTPUT TEST OK
```

BASIC equivalent:

```
PRINT"TEXT STRING OUTPUT TEST";
```

." may be used within a colon definition to print out a message on the screen when the definition is executed:

```
: ASCNUMS CR ." 0123456789" ; OK

ASCNUMS
0123456789 OK
```

CR, SPACE, and SPACES are specialty words. CR outputs a 'return' and is the same as executing PRINT by itself in BASIC. To print 3 returns in FORTH:

```
CR CR CR

 OK
```

In BASIC:

```
PRINT:PRINT:PRINT
```

SPACES prints a specified number of spaces on the screen:

```
9 SPACES          OK
```

SPACE prints just one space.

```
." ABC" SPACE ." DEF" ABC DEF  OK
```

FORTH stores character strings in memory as a string of character bytes preceeded by a byte count. To print out a character string in memory in this format, the FORTH word TYPE is used. The address of the start of the character bytes is specified, along with the byte count. Let's place a short string into memory. We'll use EBUF, which is a 30-byte buffer in FORTH used to hold the status message sent back from the disk.

(place count of characters followed by character bytes in EBUF)

```
3 EBUF C! 65 EBUF 1+ C! 49 EBUF 2+ C! 33 EBUF 3 + C!  OK

EBUF 1+ 3 TYPE  A1!  OK
```

Why did we store the byte count in the first byte when we didn't need it? To demonstrate another word, COUNT, which is given an address of the start of a FORTH string in memory. COUNT returns the starting address of the character bytes, followed by the byte count taken out of the head of the string. Since this is just what TYPE wants, then the preceeding example becomes:

```
EBUF COUNT TYPE  A1!  OK
```

You may have guessed that COUNT is defined in the FORTH dictionary as:

```
:  COUNT  DUP  1+  SWAP  C@  ;
```

To do the equivalent of COUNT TYPE in BASIC:
(assuming string & byte count start at 8000 decimal)

```
100 IF PEEK(8000) = 0 THEN 140
110 FOR X=8001 TO 8001+PEEK(8000)-1
120 PRINTCHR$(PEEK(X));
130 NEXTX
140 REM *** REST OF PROGRAM ***
```

- NUMERIC FORMATTING

Values on the stack may be easily printed. . (dot), U. , and D. have already been mentioned. These three print the number, followed by a space. Since you don't always know how many spaces the number will take up on the screen, column formatting might seem difficut. Fortunately, FORTH provides words for numeric formatting.

.R is used to print a stack value right-justified within a specified field. The number on top of the stack just as .R is executed is used as the field width. The second value on the stack is the value printed out as a SIGNED INTEGER. For instance, suppose we wish to print the top three stack values in a column, each proceeded by a message:

```
:  COLTST  CR  ." VAL1:"  5  .R  CR  ." VAL2:"  5  .R
           CR  ." VAL3:"  5  .R  ;
```

Push three values onto the stack, then execute COLTST. Each number will be printed after the message within a field 5 characters wide, right-justified:

```
21367  17  832  COLTST
VAL1:  832
VAL2:   17
VAL3:21367  OK
```

To print out a number right-justified in BASIC:

```
100 B$=STR$(S): REM **  S = NUMBER TO PRINT
110 PRINT RIGHT$("     " + RIGHT$(B$, LEN(B$)-1 ), 5)
```

U.R is used to print an UNSIGNED number right-justified, and D.R is used to print a SIGNED DOUBLE number right-justified in the same manner.

- SINGLE KEY INPUT

KEY is used to fetch the next character from a line of input. The Commodore 64 input routine is called. If the keyboard buffer is empty, the 64 waits with the flashing cursor input prompt until the RETURN key is pressed. The ASCII values of the keys in the order they were pressed are returned to the KEY routine one at a time. The value for the RETURN key (13 decimal) will indicate the end of the input line. With the way the 64 implements the input function, this routine is not of much use. The following words described will be of more help.

GET calls a C64 routine that scans the keyboard looking for a key depressed. It returns immediately, whether a key was pressed or not. The ASCII value of the key sensed is returned as the top value of the stack. If no key was depressed, then GET returns with a value of zero (0) as the top stack item. Examples of some of these input words will be given in the section LOOPING AND BRANCHING. The BASIC equivalent of GET is:

```
100 GETA$
110 IF A$=""THEN A=0 : GOTO 130
120 A= ASC(A$)
130 REM ** REST OF PROGRAM ***
```

KEYIN, like GET, is not a FORTH standard word. It is provided for convenience in this implementation. A common practice in programs is to print out a message on the screen for the user to read, and wait for a key to be pressed to continue. The trouble with doing this in BASIC is that the flashing cursor does not appear while the program is waiting for a key to be pressed:

```
100 PRINT"PRESS ANY KEY TO CONTINUE"
110 GETA$: IFA$="" THEN 110
120 REM ** REST OF PROGRAM ***
```

or:

```
100 PRINT"DO YOU WISH TO PLAY GAME AGAIN (Y/N) ?";
110 GETA$: IFA$="" THEN 110
120 IFA$="N" THEN STOP
130 IFA$<>"Y" THEN 110
140 REM ** REST OF PROGRAM ***
```

KEYIN provides the same function as line 110 of the above routines, however it also turns on the flashing cursor while it's waiting as an indication to the user.

```
: WAITKEY  CR  ." PRESS ANY KEY TO CONTINUE" KEYIN  ;
```

KEYIN turns off the cursor as soon as a key is pressesd, and returns the ASCII value of the key pressed on the stack. This value may be tested, and program branching may be based on it.

- INPUTTING A LINE OF TEXT

While KEY may be used to input a line of text from the keyboard, there is a cleaner way to do it. But before describing the commands, the description of HERE would be helpful.

HERE is a word that returns the current top of the dictionary, i.e. the address of the first free byte after the end of the dictionary. PAD returns an address that is always 68 bytes beyond the HERE address. When anything is compiled into the dictionary, thus extending it, both HERE and PAD values move upwards. The opposite happens when the size of the dictionary is decreased via FORGET.

The space from PAD downwards to HERE is used for numeric to ASCII conversion, such as in printing a stack value. The space from HERE upwards to PAD is used by the FORTH system for input stream parsing. In other words, the space starting from HERE is used as a temporary character string buffer. Each command in the input stream (that is seperated by spaces) is moved first into the buffer starting at HERE. The character string is assumed to be the name of a FORTH definition. The dictionary is searched for a match. If a match is found, the definition whose name matches the command is executed. If no match is found, the string of characters is converted into a number, if possible, and pushed onto the stack. If conversion is unsuccessful, an error message (? CAN'T FIND) is issued.

The same words that FORTH uses to scan an input line are available to the user. The buffer area starting at HERE is also available, ONLY BETWEEN COMMANDS IN THE INPUT STREAM.

The FORTH word EXPECT may be used to input a string of characters from the user, much like the BASIC:

```
100 INPUT"ENTER MONTH:";A$
```

Before executing EXPECT, the user must indicate where the string of characters is to be placed in memory. HERE is a good place, as long as you make use of all the characters input before exiting back to the FORTH input stream interpreter. Another thing that will corrupt data in the HERE buffer is doing printing of stack values, since numeric conversion starts at PAD and works downwards. This is no problem if the input string is say 40 characters or less, but is a definite problem on longer input lines. The user must also indicate to EXPECT the MAXIMUM number of characters to accept. EXPECT waits until the maximum number of characters is received, or until the RETURN key is pressed. It then puts a couple of <nulls> (0 bytes) at the end of the input string before returning. This allows the user to determine the end of the string.

```
:  ASKMONTH   CR  ." ENTER MONTH:"  HERE  9  EXPECT
              CR  ." THE MONTH ENTERED WAS:"  HERE  9  TYPE  ;
```

The character string may of course be input elsewhere in the FORTH system or moved from HERE for more permanent storage (see the CMOVE command in the glossary).

While EXPECT has it's usefulness, there is a better word. Most of the time you will need a QUALIFIER to a word. For instance, suppose you wanted to print a page header on the printer after opening it as a file. In BASIC you would do:

```
OPEN4,4,1
PRINT#4,"LISTING OF STAR SPARKYS SPACE WAR GAME"
```

FORTH has a word called WORD that will scan off a string of characters from the input stream and place them into HERE, preceeded by a byte count. The DELIMITER CHARACTER may be specified. This means that you specify the character that the text string may start and end with. This may be the space character, the quote character, or anything else.

When the word definition using WORD is executed, it must be followed by a text string that starts and ends with the specified delimiter. The file I/O commands are explained in another section, but to just explain the next example, the PRINT# routine requires the file#, address of a text string, and a text string byte count (file# addr count ---):

```
( define command to output following string to printer )

: PRNTR  34  WORD        ( SCAN OFF STRING WITHIN QUOTE CHARACTERS)
   4                     ( FILE# )
   HERE                  ( START OF STRING LEFT BY WORD )
   COUNT                 ( LEAVE ADDRESS & LENGTH OF STRING )
   PRINT#  ;             ( SEND STRING TO PRINTER )
```

Once defined, PRNTR may be used to send a character string to the printer:

```
4 4 0 0 0 OPEN
PRNTR  "LISTING OF STAR SPARKYS SPACE WAR GAME"
```

(if you tried this with a COMMODORE VIC 1525 printer, you'll have to output a RETURN to get the line to print. Type the following sequence:)

```
13  EBUF  C!  4  EBUF  1  PRINT#
4 CLOSE
```

- NUMERIC INPUT

While it is easy to push numeric values onto the stack before executing a FORTH word, it is often desirable to have a routine request a numeric value from the user. Since the user is typing alphanumeric characters, how can these be converted into numbers? In BASIC this is done automatically:

```
INPUT"ENTER STARTING NUMBER:";N
```

In FORTH, NUMBER converts a string of ASCII characters in memory into a SIGNED DOUBLE NUMBER which it leaves on the stack. The address of the FORTH character string (byte count + string) is passed to NUMBER. The number is converted according to the current numeric BASE. If a decimal point is encountered in the character string, its position is stored in the user variable DPL, but will otherwise be ignored. Since the conversion stops upon reaching a space or a null byte, EXPECT may be used to input the string as long as a byte count greater than or equal to the length of the string is placed as the first byte. An ASCII minus sign encountered at the start will leave a negative value. A general numeric input routine may now defined:

```
(  ---  d  )
: INPUT   DECIMAL                          ( or any other base )
          HERE  1+  10  EXPECT     ( read in up to 10 characters )
          10  HERE  C!               ( place byte count at start )
          HERE  NUMBER  ;             ( convert string to number )
```

To input a single precision integer, just DROP the top word off the stack after INPUT returns, or place DROP in INPUT just after NUMBER and before ; .

```
INPUT  1234  OK
.  0  OK
.  1234  OK
```

Since WORD may also be used to input numeric values, it is easy to read in and convert numeric strings representing the values in a table. For instance, in the graphics command set there is a word called SPDEFINE. SPDEFINE is used to create a sprite definition in memory. SPDEFINE calls the definition SPCODE to read in and convert 63 numbers (actually character strings representing numbers) that follow in the remainder of the screen. SPCODE is defined in the dictionary as:

```
: SPCODE
  21  0  DO                                    ( DO FOR 21 LINES )
    3  0  DO                          ( DO FOR 3 VALUES PER LINE )
      BL  WORD  NUMBER  DROP          ( READ IN & CONVERT NUMBER )
    LOOP
  SWAP  ROT       ( REVERSE ORDER OF THE 3 BYTES IN THIS LINE )
  C, C, C,            ( COMPILE THE BYTES INTO THE DICTIONARY )
  LOOP  ;
```

If a screen contains the following:

```
HEX  CREATE  SPRO  SPCODE
     00 00 00
     00 FF 00
     01 FF 80
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     03 00 C0
     01 FF 80
     00 FF 00
     00 00 00
```

When this screen is loaded, a new dictionary entry called SPRO will be created and SPCODE will read in the following 63 values and store them as bytes in the body of the definition. By modifying the above definition of SPCODE you can create varying sizes of data tables, arrays, etc.

HANDLING STRINGS IN FORTH

The techniques previously mentioned for text input and output are available in most FORTHs. C64-FORTH provides a special group of words to manipulate character strings for the convenience of the user. They are not normally part of the C64-FORTH system - they must be loaded in from disk. On your C64-FORTH disk starting on screen # 20 are the source definitions for the string words. Just type:

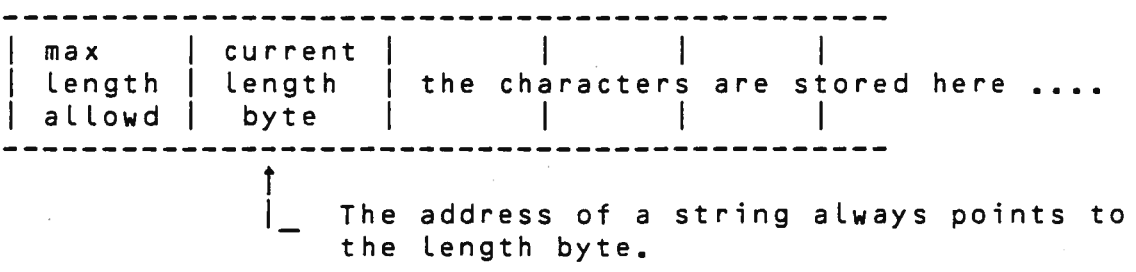
```
20 LOAD
```

and the extensions will be added onto your current FORTH system. The commands described in the rest of this chapter are now available for your use.

- HOW STRINGS ARE STORED IN MEMORY

It was previously mentioned that FORTH strings are stored as a string of characters preceeded by a length byte (note that EXPECT and TYPE are exceptions in that they deal only characters themselves, and it is up to the user to keep track of byte counts). The string extensions follow a similiar concept, however the current-length byte is preceeded by a maximum allowable length byte. The maximum allowable length for ANY string is 255 characters. Most of the string functions return an address which points to the current-length byte. Normal FORTH words such as COUNT, TYPE, CMOVE, etc. can be used to operate on strings in memory. However, when a string is stored into memory, the byte preceeding the destination is checked. If the length of the string to be stored is longer than the maximum count allowed, then only the allowed number of bytes is accepted from the string. This feature prevents the user from overwriting parts of the FORTH system by moving strings too long for the destination - a precaution most FORTHS do not provide.

A FORTH string in memory:


max length allowd
current length byte
the characters are stored here ....
The address of a string always points to the length byte.

The space starting at PAD is used as a temporary storage area for strings. If a string is currently stored starting at PAD, PAD will contain the current length of the string, the string itself will start at PAD +1 and continue upwards, and NO max byte count is stored at PAD -1. PAD is always assumed to be able to contain the maximum length for a string - 255 characters. Certain string functions also use the space from PAD+256 to PAD+511 for temporary operations, but this is transparent to the user.

The C64-FORTH extensions manipulate strings in a manner similiar to numeric stack items. If the stack has a limited depth, how can it be used to hold strings also? Easy - the strings themselves are not stored on the stack, rather the ADDRESS of the string is stored on the stack. Normal stack operators such as DUP, ROT, SWAP, etc. may be used to manipulate string pointers. This allows for very efficient string handling.

A few things must be kept in mind - One of the functions, $!, checks the specified destination address, and uses the preceeding byte as a max byte count. Therefore, you cannot use $! to move a string to some random memory location that has not been declared as a string variable or constant UNLESS you first store a max byte count into the destination address-1. You could however use CMOVE or <CMOVE to move the string there. Then any of the string functions can use that string as a SOURCE string for an operation, since the max length count is only checked when storing a string. Also, a null string is indicated ONLY by the current length byte being equal to 0 (zero). The contents of a null string may be any random pattern of byte values so only the current length = 0 should be used to detect a null string. One thing more - the address of PAD changes with the current end of the FORTH dictionary. If a string is temporarily stored at PAD, and the end of the dictionary is changed, the data starting at the new location of PAD is meaningless.

In the following descriptions, the same stack notation is used to show the stack requirements and effects as is used in the glossary. $1, $2, ... will represent strings (actually the address of the strings) on the stack.

```
-  "                    (      ---            at compile time
                               ---   $1        at run time      )
```

The first of the string extentions is " (quote). " functions similiar to ." except instead of printing a string, " moves a string into the temporary holding area at PAD. Like ." operates differently in direct (interpretive) mode than when being compiled into a definition. To put a string of characters into the PAD area, just type " followed by one space, then the string of characters delimited by another " character:

```
" ABCDEFG" OK
```

" leaves the address of PAD on the stack once it stores the string there. To see the character string as it is stored at PAD type:

```
8 DUMP          ( the address of PAD was left on the stack )
  5206   7  41  42  43  44  45  46  47    .ABCDEFG
```

(DUMP displays a section of memory starting at a specified address for a specified # of bytes. Regardless of what base you are in, DUMP always prints in HEX numbers to give a readable format on the 40-character screen. The contents of 8 locations are printed per row, followed by ASCII representations of the contents of the locations.)

" can also be used within a colon definition. When the definition is being compiled, the string is placed into the definition. When the definition is executed, the ADDRESS of the string in memory (in the body of the definition) is placed on the stack:

```
: TEST  " TEST STRING" ;  OK
```

Now type:

```
TEST  OK
U.  24872  OK
```

The address printed will be dependent upon the actual location in memory the definition TEST ended up. You may print the string by:

```
TEST  COUNT  TYPE   TEST STRING  OK
```

```
-  $.                          ( $1  ---    )
```

The COUNT-TYPE pair seems inconvenient. There are words for printing out stack values, so one is provided for printing out strings from a string address on the stack. $. prints the string pointed to by the top address on the stack. The address of the string is removed from the stack by $. From a previous example:

```
TEST  $.  TEST STRING  OK
```

— $VARIABLE (N --- at compile time
--- $1 at run time)

$VARIABLE is used to create a new dictionary entry, and reserve space for a string of a specified maximum length. When creating a new string variable, you specify the maximum length of any strings to be stored in it in the future, plus the name of the variable. For example:

16 $VARIABLE FILENAME OK

will create a string variable that can hold a string of up to 16 characters in length. The contents of the variable just created is not initialized, and may consist of random values, but the current string length is set to 0 (zero) to indicate a null string is currently stored. At any future time, just referencing the name of the string variable will push the address of the string contained onto the data stack. Examples of using string variables are given in some of the other word descriptions.

— $CONSTANT ($1 --- at compile time
--- $2 at run time)

$CONSTANT functions just like $VARIABLE except it provides a way to pre-initialize a string variable. Once the string constant is created, it appears exactly like, and can be used like, a string variable. The maximum length of the string constant defined is set to the length of the string it was initialized to. To create a string constant, first put the address of a string on the data stack, then use $CONSTANT followed by the name you wish to assign to the new constant:

" Press any key to continue..." $CONSTANT PKEY OK

" defined a string and left the address of PAD on the stack, which is where the string was temporarily stored. $CONSTANT then created a new dictionary entry, called it PKEY, and moved the string at PAD to PKEY. To verify this:

PKEY $. Press any key to continue... OK

Even though it is called a constant, you can store a string into PKEY at a future time, since it acts like a string variable once it is created.

- $! ($1 $2 ---)

$! stores string $1 at the location $2. If $2 is the address of PAD, then the maximum byte count of the destination is assumed to be 255; otherwise the byte at $2-1 is fetched and used as the max count. The MINIMUM of either the current length of $1 or the max byte count just fetched is the number of characters transferred to the destination. The length of the string formed at $2 is set to the number of characters actually moved.

From a previous example:

```
" TESTDATA.DAT" FILENAME $! OK
FILENAME $. TESTDATA.DAT OK
```

Let's try storing a string that is longer than the maximum length allowed. When creating the string variable FILENAME in a previous example, we specified a maximum length of 16 bytes or characters.

```
" ABCDEFGHIJKLMNOPQRSTUVWXYZ" FILENAME $! OK
FILENAME $. ABCDEFGHIJKLMNOP OK
```

Here it is obvious that only 16 characters were accepted when storing the specified string into a string variable of only 16 characters maximum length.

- $+ ($1 $2 --- PAD)

$+ is used to concatenate (add together) strings. $2 is added onto the end of $1 and the resulting string is left starting at PAD. Either string may be in PAD to begin with (but not both obviously), and the resulting string is left in PAD.

Examples:

```
" ABCDEF" $CONSTANT TEMP$ OK
" 123456" TEMP$ $+ OK
$. 123456ABCDEF OK

TEMP$ " 123456" $+ $. ABCDEF123456 OK
```

If the total length of the two strings combined is greater than 255 characters, an erroneous string will result.

- LEFT$ ($1 N --- PAD)

LEFT$ functions similiar to its BASIC companion. LEFT$ returns a string comprised of the left N # of characters of string $1. If the length of string $1 is less than N characters long, then the string $1 in its entirety is returned. The string $1 may be at PAD to start with, and the resulting string is left at PAD.

```
" 1234567890" 4 LEFT$ $.  1234  OK
TEMP$ 8 LEFT$ $.  ABCDEF  OK
```

Remember that when TEMP$ was defined a few examples ago, it was initialized with a string of only 6 characters, therefore the full string was returned when the leftmost 8 characters was requested.

- RIGHT$ ($1 N --- PAD)

RIGHT$ works similiar to its BASIC counterpart. The rightmost N # of characters is returned from string $1 with the resulting string left at PAD. If N is = 0, then a null string is returned, if N is greater than the length of $1, then $1 is returned.

```
TEMP$ 3 RIGHT$ $.  DEF  OK
TEMP$ 12 RIGHT$ $.  ABCDEF  OK
```

- MID$ ($1 STRTPOS #CHR --- PAD)

MID$ returns a sub-string comprised of #CHR characters starting at position STRTPOS in string $1. As in BASIC, MID$ assumes the first character in a string is at position 1, the second character is in position 2, etc... If STRTPOS is greater than the length of $1, or if either STRTPOS or #CHR is equal to 0, then a null string is returned. If string $1 has fewer characters than #CHR from the starting position to the end of the string argument, then the whole rest of the string is returned.

```
" 1234567890" 3 4 MID$ $.    3456  OK
" 1234567890" 10 1 MID$ $.    0  OK
" 1234567890" 12 3 MID$ $.    OK      (null string was
                                       returned)
```

- LEN ($1 --- LEN)

LEN returns the current length of the specified string. To use a previous example:

```
TEMP$ LEN . 6 OK
" 1234567890" 5 0 MID$ LEN . 0 OK   (a null string was
                                      returned by MID$)
```

- CHR$ (B --- PAD)

CHR$ returns a 1-character string created from the byte value passed on the stack. This may be used to insert carriage-returns in a string, printer control commands, etc.

```
TEMP$ 13 CHR$ $+ $. ABCDEF
 OK
```

- ASC ($1 --- B)

ASC returns a number from 0-255 which corresponds to the Commodore ASCII value of the first character in the specified string. If there are no characters in the string, a "NULL STRING" message will be printed and a value of zero will be returned. The null-string message can be avoided by adding (concatenating) a string equal to CHR$(0) onto the end of the string to use ASC on.

```
" ABCDEF" ASC . 65 OK
" " ASC . NULL STRING 0 OK
0 CHR$ TEMP$ $! " " TEMP$ $+ ASC . 0 OK
```

- VAL ($1 --- N)

VAL converts a string of characters into a single-precision number and returns it on the stack. The conversion stops at the first non-digit character. The character string may start with a minus sign, and the converson is done according to the current base.

```
" -1234" VAL . -1234 OK
" 65530" TEMP$ $! TEMP$ VAL . -6 OK
```

- DVAL ($1 --- D)

DVAL works just like VAL except a double-precision # is returned:

```
" 12345678" TEMP$ $! OK
TEMP$ DVAL D. 12345678 OK

HEX TEMP$ DVAL DECIMAL D. 305419896 OK
```

Note that in the last example the conversion pushed a value equal to 12345678 HEX onto the stack, which when printed in decimal comes out 305419896.

- STR$ (N --- PAD)

STR$ takes the top stack value and converts it to a string of characters, leaving it at PAD. The resulting string begins with either a space or minus sign depending on the sign of the original value.

```
1234 STR$ 8 DUMP
  6592   5  20  31  32  33  34  20  20   . 1234     OK

-32760 STR$ 8 DUMP
  6592   6  2D  33  32  37  36  30  20   .-32760    OK
```

- DSTR$ (D --- PAD)

DSTR$ acts the same as STR$ except a double-precision number is taken from the stack and converted to a string. No decimal point is included in the string, however a definition can be created for that purpose :

```
12 $VARIABLE S1  12 $VARIABLE S2    ( temporary variables )
: DLRSTR$                           ( D --- PAD )
  DSTR$ " $" S1 $+ S1 $!
  S1 DUP LEN DPL @ - LEFT$ S2 $!
  S1 DPL @ RIGHT$ S1 $!
  S2 " ." $+ S1 $+ ;                ( string left at PAD )
```

DLRSTR$ may now be used to create a dollar value out of a double number on the stack (note: the position of the decimal point must be in variable DPL, which it already is if the double number on the stack has been the most recent number converted from the input stream) :

```
534.86  DLRSTR$  $.  $ 534.86  OK
```

The DPL @ pair used in DLRSTR$ may be replaced by the constant 2 to ignore the position of the decimal point in the number input to always have 2 decimal places in the final string. S1 2 255 MID$ S1 $! may be used after DSTR$ in DLRSTR$ to eliminate the leading space thus putting the dollar sign right before the value, and other improvements may be made in a similiar manner.

- INPUT$ ($1 ---)

INPUT$ provides very convenient way to input a character string and store it in a variable. In BASIC we would do :

```
10 INPUT A$
```

INPUT$ allows us to do a similiar thing in FORTH. The FORTH word EXPECT is called, which requires a maximum count of characters. If the address to store the string being input ($1) is equal to PAD, then 255 is used as the input count; otherwise the maximum length of the destination is used.

```
80  $VARIABLE  A$
:  GETDATE  CR  ." INPUT TODAY'S DATE: "  A$  INPUT$  ;
```

When GETDATE is executed, a character string of up to 80 characters (the maximum length of string variable A$) will be accepted and stored in A$.

```
GETDATE
INPUT TODAY'S DATE:  JULY 27, 1983   OK
A$  $.  JULY 27, 1983  OK
```

- POS$ ($1 $2 --- N)

POS$ searches for the first occurance of string $2 within string $1. If found, N is returned which is the position of the start of the matching string within $1. If not found, N returns equal to zero. Using the string input into A$ in the previous example:

```
A$  " 27,"  POS$  .  6  OK
A$  " 1983A"  POS$  .  0  OK      ( no match found )
A$  " JULY"  POS$  .  1  OK
```

- $= ($1 $2 --- flag)

$= is used to compare two strings and returns a true flag if the strings contain the same characters AND they are equal in length.

```
" ABCDEFGH" A$ $! OK
" ABCDEF" A$ $= . 0 OK   ( strings not same length )
" ABCDEFGH" A$ $= . 1 OK    ( strings identical   )
```

- $< ($1 $2 --- flag)

$< compares $1 to $2 and returns a true flag if $1 is less than $2, OR if $1 is shorter than $2.

```
" ABCDEF" A$ $< . 1 OK   ( $1 is shorter than $2 )
" ABCDEFGH" A$ $< . 0 OK  ( $1 is not less than $2 )
" ABCDEFGG" A$ $< . 1 OK  ( $1 is less than $2 by values)
```

- $> ($1 $2 --- flag)

$> compares $1 to $2 and returns a true flag if $1 is greater than $2, OR if $1 is longer in length.

- $NUMBER ($1 --- D)

$NUMBER functions almost identical to the FORTH standard word NUMBER. However, NUMBER aborts with an error if the string being converted into a double number contains an unexpected character. $NUMBER converts a string of characters into a double-precision stack value, but only stops the conversion process upon encountering a non-expected character in the string. 'Expected' characters means a possible leading minus sign, an embedded decimal point, and any numeric value legitimate to the current base. The address of the first non-expected character within string $1 that the conversion stops at is left in the variable $NUMADR.

$NUMBER will prove very useful at times. If you have created a data file from within a BASIC program, and PRINTed numeric values out to it, you will have a data file containing something like this on disk:

```
34, 567,JOHN SMITH,1428 ELM ST.,LIVELYHOOD,KA,999-555-8426
```

The above may have been created with a BASIC statement such as:

```
2020 PRINT N;B;N$;T$;C$;S$;P$
```

BASIC seperates PRINTed values by commas, which are not 'expected' characters in NUMBER conversion, but $NUMBER will convert one of the numbers in the above example and stop the conversion process upon encountering a comma. The address of the character that stopped the conversion, in this case one of the commas, is left in $NUMADR. A combination of the value left in $NUMADR, and proper use of MID$ can be used to scan off and convert a data file saved on disk via either BASIC or C64-FORTH.

OTHER DEFINITIONS:

Following are a few other definitions used in the string extension screens that may occasionally be useful:

- -TEXT (addr1 N1 addr2 --- N2)

-TEXT is a PROPOSED FORTH STD word. The description of the proposal (and the operation of the word as defined in the C64-FORTH string extensions) is: "Compare two strings over the length N1 beginning at addr1 and addr2. Return zero if the strings are equal. If unequal, return N2, the difference between the last characters compared: addr1(i) - addr2(i) ".

Note that the addresses passed to -TEXT are NOT normal string address (i.e. of the byte count at the start of a string), but rather absolute addresses within strings of characters.

- $COMPARE ($1 $2 --- flag)

Compares $1 with $2 returning a flag as follows:

1) Positive if $1 is greater than $2 OR equal to $2 but longer,
2) Zero if the strings are identical in content and length,
3) Negative if $1 is less than $2 OR equal to $2 but shorter.

- UMOVE (fromadr toadr #byt ---)

UMOVE is a universal version of CMOVE and <CMOVE. UMOVE decides which version to call, taking into consideration source-destination overlap.

FILE I/O

The FORTH screens generated by C64-FORTH are saved onto disk or tape as sequential data files. The routines used by C64-FORTH to do this are all available to the user. They allow creating or accessing FORTH screens as data files, and data files created by or to be used by BASIC programs. As in BASIC, tape I/O is as easily worked with as disk I/O. Program type files and relative files also are accessable.

The one major difference between accessing data files via C64-FORTH and through BASIC is that C64-FORTH does data reading and writing from/to an external file on a straight byte-by-byte basis. It is up to the user to determine and identify whether data is numeric, components of ASCII string data, etc. Conversely, error checking is easier than possible in BASIC.

- SYSDEV#

SYSDEV# is a C64-FORTH variable used by the I/O routines. Most routines automatically set it to the device number of the I/O unit currently being accessed. The low-level routines of OPEN, CLOSE, GET#, INPUT#, PRINT#, AND CMD# do not use SYSDEV# at all, they rely upon the device number being passed through the OPEN command. However, some of the disk error checking routines do require that the device number of the disk be present in SYSDEV#. It is a good practice to set SYSDEV# before opening any files to a disk or tape device. Note that the low byte of the contents of SYSDEV# hold the device #, and the high byte, normally 0, indicates which drive to use of a multiple-drive unit such as a 4040. Also, SAVESYSTEM and SAVETURNKEY commands require that SYSDEV# be holding the device number of the I/O unit that C64-FORTH is to be saved on.

- ST

ST is a C64-FORTH variable. After any I/O using OPEN, CLOSE, GET#, INPUT#, and PRINT#, the ending I/O status is left in ST. In BASIC programs, the variable ST is usually checked after a transfer:

```
100 INPUT#2, A$(I),B          : REM ** READ IN STRING & NUMBER
110 IF ST<>0 THEN 1000        ; REM ** LINE 100 IS ERROR ROUTINE
```

The same can be done in C64-FORTH:

```
 ...  2  HERE  80  INPUT#  ST  C@  IF  ." I/O STATUS ERROR"  ...
```

```
- OPEN        ( file#  dev#  sa  fnptr  fnbytcnt  ---  )
```

A file must be opened before it may be accessed. Parameters must be passed to the OPEN command giving the specifications necessary to access the file. The major difference between the FORTH OPEN and the BASIC OPEN commands is that with BASIC OPEN, an optional filename or command string may be specified, whereas the FORTH OPEN must be passed a filename/command string count and pointer EVEN IF ONE DOES NOT EXIST (in this case a 0 is pushed onto the stack for a filename/command string count, and any value may be pushed on as a filename/command string pointer, as this will be ignored).

```
( assuming a filename/command string and byte count is in a data
type called FNBUF and FNBUF contains the byte count of 14
followed by the string '0:TESTDATA,S,R' )

2               ( push a file# onto stack )
8               ( push device # onto stack )
2               ( push secondary address onto stack )
FNBUF  COUNT    ( push filename string address and byte count )
OPEN            ( open the device for input )

BASIC equiv:
OPEN 2,8,2,"0:TESTDATA,S,R"

( to open a device without specifying a filename or command
string )

4               ( push file# onto stack )
4               ( push device # )
1               ( push secondary address )
0               ( push any value to keep stack balanced )
DUP             ( push filename byte count - 0 in this case )
OPEN            ( open the device )

BASIC equiv:
OPEN 4,4,1



- CLOSE       ( file#  ---  )
```

Any file that was opened must be properly closed to ensure all data is written to the file, and to free up the file or device for further use.

```
2  CLOSE       ( close file # 2 )

BASIC:
CLOSE 2
```

- INPUT# (file# addr count ---)

This is the easiest way to retreive character data from a file on disk or tape. The count specifies a MAXIMUM number of characters to input (BASIC automatically limits at 80). INPUT# terminates before reaching the maximum count if a RETURN (byte value =13), a <null> code (byte value =0), or the END-OF-FILE status is received. The data received, including the RETURN or NULL byte if encountered, is placed in memory starting at <addr+1>. The count of the actual number of bytes received (this does not include the RETURN or NULL bytes) is placed at <addr>, forming a normal FORTH string.

```
2  HERE  80  INPUT#     ( read 80 characters max. into memory
                          starting at end of dictionary )
```

```
BASIC:
INPUT#2,A$ or INPUT#2,C
```

- GET# (file# addr count ---)

Similar to INPUT#, GET# retrieves <count> number of bytes from <file#> and stores them as a FORTH string (byte count + bytes received) starting at <addr>. Input is terminated before reaching maximum count ONLY by detection of the END-OF-FILE status. GET# can be used instead of INPUT# when the input data is expected to contain byte values equal to 0 or 13 decimal. GET# is more useful than BASIC's GET# because it can be used to input more than one byte at a time. Since the setup for a transfer of a byte is far greater than actually transferring the byte, multiple byte GETs will be very fast, compared to putting GET# in a loop in BASIC.

```
( fetch 20 bytes from data file )

7  HERE  20  GET#
```

```
In BASIC:
100 FOR Y=1 TO 20
110 GET#2,A$ : A$=A$+CHR$(0)
120 A = ASC(A$)
130 POKE AD,A
140 NEXTY
```

- PRINT# (file# addr count ---)

PRINT# is used to transfer data bytes to an external file. <count> number of bytes are transferred starting from <addr>. If a FORTH string starts at HERE, then to send it to file# 5:

```
5  HERE  COUNT  PRINT#
```

This is similar to BASIC's: PRINT#5,A$

NOTE: 5 0 0 PRINT# is NOT identical to BASIC's PRINT#5 by itself, since Commodore 64 BASIC automatically outputs a RETURN after a print statement that doesn't end with a semi-colon. To do the identical function, place a RETURN code somewhere in memory, and output that:

```
13  EBUF  C!  5  EBUF  1  PRINT#
```

- CMD# (file# ---)

CMD# is similar to BASIC's CMD with a slight difference (hence the different name); output is not diverted from the screen to the external device, but is sent out seperately after it is printed on the screen. In other words, the external device is opened, a byte transmitted, and then closed each time a byte is printed on the C64 display. Since disk versions of FORTH rely heavily on the disk, this was necessary. Otherwise, to list multiple screens for instance, a screen would have to be read in, the CMD file opened, CMD issued, the screen printed, the CMD device closed, the next screen read in from disk, etc. Disk directories would have to be buffered in memory. CMD# implements this in a much cleaner way allowing true hardcopy of the screen display.

For an example of usage, see the section under 'Details of the C64-FORTH system' called 'Listing multiple screens on the VIC-1525 printer'.

- ABORTIO

This routine may be called from any FORTH routine. It closes ALL open files. This may be used in case of an I/O error.

- GETDEV (file# --- dev#)

GETDEV returns the device number that a specified file number was opened with. The file number must be a currently open file.

- EBUF (--- adr)

EBUF is a 30-byte long buffer used to hold the disk status sent from the disk. See RDSTAT, CKRDSTAT, and CKWRSTAT.

- RDSTAT (file# ---)

RDSTAT reads in the disk status from the disk ONLY if file# refers to a physical device number > 3. It is up to the user NOT to use this routine on any device that is not a disk or cassette. The status read in from the disk is left in EBUF as a FORTH string. If the device is a cassette (i.e. the device # < 4), then an ASCII "00" (zero-zero) is left at EBUF+1 and +2. Since the first two ASCII bytes returned as a disk status indicates the error code, a FORTH routine may be generalized with regard to disk or cassette. RDSTATting from a cassette will return an 'OK' status (indicated by the two ASCII zeros).

- CKST (--- flag flag=0 if ok, =1 if end-of-file)

CKST can be used to check the I/O status after a transfer (see ST). If the status byte was equal to 0, a FALSE flag is returned. If an END-OF-FILE is detected, this is not usually an error but an indication for the program, hence CKST will return TRUE. Any error in the status, and CKST does not return. It prints the error message 'I/O STATUS ERROR: ' and prints the status byte in the current BASE. CKST then returns execution to the FORTH input interpreter.

- CKRDSTAT (--- flag flag=0 if ok, 1 if end-of-file)

CKRDSTAT first calls CKST to check the I/O status byte. Then it reads the ERROR CHANNEL to the disk (OPENCHN opens the error channel as file# 15). CKRDSTAT returns FALSE if everything is ok, TRUE if end-of-file was detected. If the disk status indicates an error (i.e., an error code >=20 is returned), then the error message received from the disk (and stored in EBUF) is printed. All open I/O files are closed, and the C64-FORTH disk error message is printed (ERROR is executed). Control returns to the FORTH input interpreter.

- CKWRSTAT (---)

CKWRSTAT is identical to CKRDSTAT except no flag is returned. This is not necessary when checking disk status after WRITING to the disk, since the end-of-file indication is meaningless and will never occur. CKWRSTAT can be used to check disk status when reading, if you know the end-of-file status will not occur, or you expect it to occur reading the last byte of a file of known length.

- ?DISK (--- flag flag=0 if not disk, <>0 if disk)

?DISK returns a flag based on whether the device number in SYSDEV# is a disk or not. The only determination made is whether the device number is >= 4.

- OPENCHN (---)

OPENCHN opens the error channel to the disk. OPENCHN is defined as:

```
: OPENCHN ?DISK IF 15 SYSDEV# C@ 15 OPEN CKWRSTAT THEN ;
```

In BASIC, this would be:

```
10 OPEN 15,8,15
20 IF ST<>0 THEN PRINT"I/O STATUS ERROR:";ST : STOP
30 GOSUB 1000
1000 INPUT#15,EN,EM$,ET,ES
1010 IF EN=0 THEN RETURN
1020 PRINT"DISK ERROR ",EN;EM$;ET;ES : STOP
```

- CLOSCHN (---)

CLOSCHN closes the error channel to the disk that was opened by OPENCHN. CLOSCHN is defined as:

```
: CLOSCHN ?DISK IF 15 CLOSE THEN ;
```

```
- ?FILE    (  ---   flag      flag=0 if ok, =1 if file not found )
```

?FILE is handy after opening a file to see if it exists on disk, or if the 'file does not exist' code is returned by the disk. ?FILE is defined as:

```
HEX  :  ?FILE  CKST  DROP  ?DISK  IF
          15  RDSTAT  EBUF  1+  @  DUP  3236  =  IF
             DROP  1  ELSE  3030  -  IF
                CR  EBUF  COUNT  TYPE
                CR  ABORTIO  8  ERROR
             THEN
          ELSE  0  THEN  ;
```

```
*********************************
*****   SAMPLE PROGRAMS   *****
*********************************
```

This is a good spot to give some sample programs. All three of the following examples require the C64-FORTH string extensions.

#1 - Sending a command to the disk

The C64-FORTH word DC can be used to send a command to the disk, however, it cannot be compiled into a definition. To define a version that can be compiled into a definition:

```
:  DCMD                                  (  $adr  ---  )
( DCMD sends the string at the address on the stack to the disk )
( Note: do not use this to send the directory command          )
     15  SWAP  COUNT  PRINT#             ( send string )
     CKWRSTAT  ;                         ( check disk status )
```

Now, lets say we happen to have a collection of files on the disk, called 'CLIENT1', 'CLIENT2', 'CLIENT3', 'CLIENT4', and 'CLIENT5'. We had written a simple filing system, which had left these temporary files out on the disk. Each one contains all the data relating to one client account. We have dropped client #1, so we wish to drop his data, and consider client #2 to be the new client 1, etc. The following routine will delete CLIENT1, and rename the subsequent files.

```
20  $VARIABLE  N$                       ( define temporary string var. )
:  REMOVECLIENT1
     OPENCHN                            ( open error channel to disk   )
     " S0:CLIENT1"  DCMD                ( remove client 1 file )
     6  2  DO                           ( set up DO LOOP to rename files )
       " R0:CLIENT" N$ $!               ( form rename command string )
       I STR$ 2 255 MID$ N$ SWAP $+ N$ $!
       N$ " =CLIENT" $+ N$ $! N$ I 1+ STR$ 2 255 MID$ $+
       DCMD                             ( send command string to disk )
     LOOP
     CLOSCHN   ;                        ( close error channel )
```

#2 - Print the starting and ending load addresses of a PRG file

The following routine requests the name of a file on the disk in dirve #8, then prints out the starting and ending load addresses of that file.

```
16 $VARIABLE N$                    ( temporary string variable )
: LOADADR
  CR ." INPUT FILENAME: "          ( get name of file from user )
  N$ INPUT$
  CR ." LOAD ADDRESSES OF " N$ $. ." ARE..." CR
  OPENCHN                          ( open error channel )
  14 8 14 N$ " ,P,R" $+ COUNT OPEN      ( open the file )
  ?FILE IF                         ( does the file exist ? )
    CR ." FILE DOES NOT EXIST"           ( no )
  ELSE
    HEX 14 HERE 2 GET#             ( yes, get starting load address )
    CR ." STARTING ADR: "          ( and print it )
    HERE 1+ @ DUP 4 U.R CR         ( leave address on stack )
    BEGIN                          ( read in all bytes in the file )
      14 HERE 1 GET#               ( one at a time and increment the )
      1+                           ( address for each byte )
    CKST  UNTIL                    ( end-of-file ? )
    1-                             ( yes, back address up one )
    ." LAST BYTE ADDR: " 4 U.R              ( and print it )
  THEN
  CR 14 CLOSE CLOSCHN DECIMAL ;          ( close channels )
```

#3 - Copy a screen the hard way - a 1024 byte file-to-file copy.

The following routine copies the contents of one C64-FORTH screen to another. While the C64-FORTH command COPY will do this nice and easy, this is a simple example of data I/O to and from the disk.

```
3 $VARIABLE A$  3 $VARIABLE B$  10 $VARIABLE C$
: CKFILE            ( ---  flag    flag =0 if file wasn't found)
  ?FILE IF
    CR ." FILE CALLED " PAD $. ." NOT FOUND" 0
  ELSE  1  THEN  ;

: COPYSCR                              (  ---  )
  CR ." SCREEN # TO COPY FROM: "  A$  INPUT$
  CR ." SCREEN # TO COPY TO  : "  B$  INPUT$
  OPENCHN
  " SCR" A$ $+ C$ $!
  14 8 14 C$ " ,S,R" $+ COUNT OPEN       ( open input file )
  CKFILE IF                              ( does file exist ? )
    " SCR" B$ $+ C$ $!
    13 8 13 C$ " ,S,W" $+ COUNT OPEN     ( yes, open output file )
    CKWRSTAT
    14 HERE    1024 GET#    CKWRSTAT     ( read in 1K of data )
    13 HERE 1+ 1024 PRINT#  CKWRSTAT     ( write it out to file )
  THEN
  13 CLOSE 14 CLOSE CLOSCHN CR ;         ( close all files )
```

- SAVESYSTEM and DC

Every FORTH user develops his or her own custom word definitions to do common functions easily. The desire to make them a permanent part of the FORTH system soon arises. Instead of booting up the original version of C64-FORTH each session and loading in the screens with the definitions, it is very easy to save the current system with all new additions onto disk. The SAVESYSTEM command is used as follows:

```
SAVESYSTEM  "MYFORTH"
```

SAVESYSTEM first prompts the user to insert the ORIGINAL C64-FORTH disk into the drive. SAVESYSTEM reads a section of it, then prompts the user to insert the disk the FORTH system is to be saved on.

The current version of the FORTH system will be saved onto disk as a file named "MYFORTH" (or whatever filename you put between the quotes). The next time you power-up the system just load "MYFORTH" (or whatever you called it), and RUN. Do a VLIST and you will see that your custom definitions are part of the FORTH system.

The string between the quotes is sent directly to the disk or cassette, so disk file replace functions work:

```
SAVESYSTEM  "@0:FORTH2.EXE"
```

C64-FORTH provides a word to send commands to the disk when in the interpretive mode. DC sends whatever is between the quotes following it to the disk, and reads the status back. DC can be used alone without a following string to just read back and check the status. The status is not printed if it has a status value less than 20. In the current 1541's, the only two status codes returned less than 20 are 'OK' and 'FILES SCRATCHED'. Examples:

```
DC  "N0:FORTHWORK,01"        ( send cmd to format a disk )
DC  "S0:TEMP.DAT"            ( scratch file )
DC                           ( just read and check status )
```

The user should never use the original C64-FORTH disk, except to load C64-FORTH for the first time, and for SAVESYSTEMing and SAVETURNKEYing. By using the DC command along with SAVESYSTEM, you can easily format and save the FORTH system off onto a new disk. A disk with the C64-FORTH program on it appears as any other Commodore-64 program disk. Screens are saved as ASCII sequential data files. This means you can save other kinds of programs as well as data files created by BASIC programs, relative files, etc. By changing just a variable or two, C64-FORTH can be quickly adapted to make full use of an extended capacity floppy or Winchester drive.

Screens may be moved from one disk to another by writing a routine to read them in from one disk (it would help to allocate more disk buffers via the BUFFERS command), mark them as updated, then place the new disk in the drive and execute SAVE-BUFFERS.

There is no initialization of screens necessary for a NEW'ed blank disk. If a screen or block is requested, and the specified screen does not exist on disk, a screen full of blanks is delivered to the user. Only if the blank screen is altered or marked as updated, AND SAVE-BUFFERS is executed, will it be placed on the disk. (See the description of TAPEFLG for how this works with cassette)

- DIR

DIR displays the directory of drive 0 of the disk specified in SYSDEV#. DC "$" also calls the routine DIR. DC "$1" may be used for the directory of drive 1 of a multiple-drive disk. The CTRL key may be used to slow down the listing, and the STOP key may be used to terminate the directory command. DIR may ONLY be used in the interpretive mode (i.e. cannot be compiled into a definition).

- LISTING MULTIPLE SCREENS ON THE VIC-1525 PRINTER

To list more than one screen, the following word will provide proper page formatting on a VIC-1525 printer:

```
: LS 1+ SWAP DO I LIST CR CR CR CR LOOP ;
```

To list screens 60-79 on the printer:

```
4  4  0  0  0  OPEN
4  CMD#
60  79  LS
4  CLOSE
```

- BUFFERS, MEMTOP, AND USING AVAILABLE RAM

When C64-FORTH first starts running, it maps (switches) out the BASIC ROMs, and assumes the top of usable memory to be CFFF. It first allocates 512 bytes below this point for use as RS-232 buffers, should they be needed. Just below this, 128 bytes is allocated as the user area. The user area is actually 64 16-bit storage locations, used to hold system parameters of the current C64-FORTH system. Entries in the FORTH dictionary called User Variables store their contents here, not in the body of the definition in the dictionary.

C64-FORTH then allocates block I/O buffers below the user area. Four buffers are immediately allocated. The number of buffers may be any number from 2 up to what space permits. Each buffer takes up 1028 bytes. The command BUFFERS saves any updated screens, multiplies the desired number of buffers by 1028, checks if there is enough room between HERE+200 and the bottom of the user area (LIMIT returns that address). If there is, the specified number of buffers is reserved, and FIRST shows the new bottom of the buffer area. If not, the number of buffers that there is room for is reserved.

```
8 BUFFERS      ( reserve 8 block buffers, if there is
                 enough room for them )
```

It is important to note something here - Since PAD always points to a location 68 bytes above the current location of HERE (i.e., the end of the dictionary), and the C64-FORTH string extensions use the area from PAD to PAD+512, then you cannot use the string extensions if you have the maximum number of block buffers allocated that space permits. Also, if there is not enough space to allocate the requested number of block buffers, the user is not informed of how many buffers actually were allocated. However, using the values in FIRST and LIMIT and the value of 1028 bytes per buffer, it is easy to calculate the actual number allocated.

MEMTOP is used to lower the effective top-of-memory of the FORTH system. This is helpful in reserving a section of RAM that FORTH will not touch except by commands from the user. The RS-232 buffers and the user area will be relocated just below the address specified, and MEMTOP will attempt to allocate 4 block buffers just below the user area. In deciding how far you can move the top-of-memory down, check the current top of the dictionary (returned by HERE), add 840 to it (200+512+128), and add the number of buffers desired times 1028. Subtract the result from the physical top of memory (D000 hex). This will give you the maximum number of bytes that you may move the top down.

Example:

HERE 840 + HEX U. 60F2 OK (HERE + 200 + 128 + 512)

404 2 * 60F2 + U. 68FA OK (we only need 2 buffers)

D000 68FA - U. 6706 OK

(6706 hex is approx 26.4K bytes free. We'll reserve just 16K bytes. This will leave 10K free for adding new definitions.)

D000 4000 - MEMTOP (MEMTOP executes COLD when done)
C64-FORTH

2 BUFFERS OK

(prove that everything is lower)

HEX LIMIT U. 8D80 OK

FIRST U. 8578 OK

What if at run-time you would like to reserve some space to use as a buffer, but you are not sure which RAM locations between the top of the dictionary and the bottom of the buffers are safe to use? The best way is to request one or more of the block buffers. The BUFFER command allocates a 1K byte block of memory and assigns it to the block number specified. The previous contents of the buffer, if updated, are written out to disk. The new block is NOT read in. This is not safe to do if your program will be reading in blocks (screens). But for application programs where no disk or cassette I/O is done except for data file reading or writing, all allocated buffers are free to use.

A block number is passed to the BUFFER command. It's best to pick a block number that is not configured on any I/O storage unit. 7FFF is a good buffer number to request since it's meaningless. Just don't use a negative number. BUFFER returns the address of the first location in the 1K of space assigned.

Example:
HEX 7FFF BUFFER U. C376 OK

To create a table, array, or machine code overlay and save it on disk for future use, obtain use of a buffer as described above, except specify the screen number you want it stored into on disk. Immediately after obtaining the buffer, execute the UPDATE command to have that buffer marked as updated. Fill it, then execute SAVE-BUFFERS to have it written out to disk. To read in the screen at some future time, specify the screen number and execute BLOCK. BLOCK calls BUFFER to request space to read in a screen. The screen is read in and BLOCK returns the address of the first byte in the block.

Example:
(save a 1K table of data onto disk screen #20)

```
20  BUFFER          ( request space for block #20 - addr returned)

( execute code here that fills buffer in memory )

UPDATE  SAVE-BUFFERS      ( write block out to disk screen #20 )
```

- BORDER, BKGRND, AND CHRCLR VARIABLES

When C64-FORTH is entered from BASIC or when either COLD or WARM is executed, the colors of the C64 border, background, and characters printed are set from the color codes in these variables. These may be changed at any time. The color codes must be in the range 0-15.

- RESTORE KEY

The RESTORE key on the C64 keyboard, when pressed with the STOP key, can be used to restart the FORTH system, in case it 'hangs' up. This is effectively a 'warm start', meaning that definitions created since boot-up are not l

RESTORE will not work if the FORTH kernal itself has somehow been corrupted, or if the interrupts were turned off when setting up special graphics modes in the VIC-II chip. Data or return stack overflow can also corrupt the system to where recovery is impossible. Upon boot-up, C64-FORTH writes a short NMI handler into the RAM underneath the kernal ROM. This is to allow proper NMI handling via the RESTORE function even if high-res graphics are currently being drawn to the RAM underneath the kernal ROM, in which case the kernal ROM is temporarily mapped out. If the kernal ROM is mapped out at the instant of a NMI interrupt, then only a RESTORE key generated interrupt is recognized. A NMI interrupt from a RS-232 channel input or output operation is disregarded, therefore RS-232 channel I/O CANNOT be enabled during graphics writing to RAM underneath the kernal ROM.

- FUNCTION OF DV1, DV2, DV3, AND DV4

Up to four storage devices may be easily configured into the C64-FORTH system. Four variables exist that define the characteristics of storage devices available, named DV1, DV2, DV3, and DV4. The variables each contain 4-bytes of data. The first two bytes contain the number of screens that device may hold. The third byte in each specifies the I/O device number of that device (only 1 or 4-15 is valid as a device number). The fourth byte contains the drive number of the unit. Normally zero, this can be set to 1 to access drive #1 for 4040 dual-drive type disk units attached via an IEEE cartridge.

When the number of screens in one of the variables is zero, then that I/O unit, as well as the following units, is not available for storage of screens. The number of screens a device is allowed to store is found by reading the first 2-bytes of its variable:

```
DV1  @  U.      ( will print out the # of screens/blocks allowed
                  on memory device #1 )
```

You may change the number of screens allowed on a memory device (or to enable a new device if it is attached to either the serial bus or via an IEEE interface) by storing a new number in the device variable. For instance, you have just hooked up another 1541 drive (the internal jumpers were changed to configure it as device #9), and you want to use it to store screens. The first 1541, device #8, is characterized by DV1. We must now characterize DV2 to allow C64-FORTH to recognize the new drive. We must inform C64-FORTH of the number of screens, the bus device #, and the drive #:

```
120  DV2  !  9  DV2  2+  C!  0  DV2  3  +  C!
```

Each screen takes up 5 disk blocks. A screen takes up space on disk ONLY when a screen has been stored to the disk. But assuming the full number of allowable screens are stored on each disk, 120 is the maximum practical amount storable on a 1541. If you intend to have other programs stored an a screen disk at the same time, you must lower the number of allowable screens to compensate. For convenience, lets make sure DV1 is set to 100 (C64-FORTH is shipped with DV1 set to 100, but lets make sure):

```
100  DV1  !
```

And to make sure C64-FORTH doesn't think there are any more devices available, we will store a zero into DV3 and DV4:

```
0  DV3  !  0  DV4  !
```

Next we should set the device and drive number of the new drive:

```
9  DV2  2+  C!  0  DV2  3  +  C!
```

DV1 should already have 8 and 0 stored as the device # and drive #. What we have just done is configure the C64-FORTH system as follows:

- Memory device #1 is bus device #8, drive #0, and holds FORTH screens # 0 to 99. The screens are stored on the disk currently in drive 0 of device #8 as files with names SCR0 thru SCR99.

- Memory device #2 is bus device #9, drive #0, and holds FORTH screens # 100 to 219. The screens are stored on the disk currently in drive 0 of device #9 as files with names SCR0 thru SCR119.

Note that the file names of screens stored on a disk are relative to SCR0. This means that if, referring to the above configuration as an example, you store a screen as screen # 123, it will be saved on drive #0 of device #9 as SCR23. You can at a later time put the disk that was in device #9 into device #8, and access that same screen as screen #23. The way the algorithm works is:

- Screens are saved on memory device #2 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory device #1 (i.e. the contents of DV1 are subtracted from the requested screen #).

- Screens are saved on memory device #3 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory devices #1 and #2 (i.e. the contents of both DV1 and DV2 are subtracted from the requested screen #).

- Screens are saved on memory device #4 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory devices #1, 2, and 3 (i.e. the contents of DV1, DV2, and DV3 are subtracted from the requested screen #).

Commands such as LIST, EDIT, BLOCK, and COPY work on any screens on any configured disk, as long as the requested screen # is within range. If not, an error message will be displayed.

The cassette unit may be used as freely as a disk drive for storing screens. Now since the FORTH will return a screen full of blanks if a requested screen does not already exist on the disk, how does C64-FORTH know that a screen does not already exist on a cassette? There is no way of it knowing. A variable called TAPEFLG has been added for that purpose, and is described elsewhere in this section.

- FUNCTION OF SYSDEV#

SYSDEV# is a variable used by the C64-FORTH I/O routines. In any case where a BLOCK is read in or transferred out, R/W automatically sets this variable from the values in DV1, DV2, DV3, and DV4. But when a SAVESYSTEM or SAVETURNKEY is executed, the program is saved onto whatever device number happens to be in SYSDEV#. This is no problem if the disk drive that C64-FORTH was loaded from is the one you wish to save the current FORTH system to. If not, or if you've done block I/O (loading or editing screens for instance) from a different device, then you must set SYSDEV# before executing a SAVESYSTEM or SAVETURNKEY. The low byte of SYSDEV# holds the device #, and the high byte holds the drive #.

Example:

```
8  SYSDEV#  !  SAVESYSTEM  "CURRENT SYS"
```

- TAPEFLG

TAPEFLG is a variable used for cassette block I/O. The problem arises when you wish to edit a screen that does not already exist on the cassette. With the disk, no problem. R/W detects the FILE DOES NOT EXIST status from the disk, and creates a blank screen in memory to start with. But since the cassette cannot return that status, TAPEFLG was created to preserve the structure of the R/W routine.

If the screen number we wish to edit does not exist on cassette, the FORTH system could look forever for it and never find it. So when a screen is to be read in, and the device that screen is associated with is the datasette (i.e. device #=1), then TAPEFLG is checked. If TAPEFLG is =0, then the cassette is searched for the screen. If TAPEFLG >0 and <256, then an inquiry is printed on the screen asking whether it should look for the file on the tape, or should create a new blank screen in memory and assign it to the screen number specified. Any reply, including just hitting the RETURN key, will cause searching for the file on the cassette. Only by pressing the 'N' key will a blank screen be created.

- VALIDATING THE SCREENS DISK, AND DISK PECULIARITIES

FORTH screens are saved onto the disk as sequential data files. When a screen is to be saved out to the disk, the data is saved in the REPLACE mode. This means the filename the screen was saved out under was preceeded by "@0:". This causes no problems most of the time, but occasionally a nearly-full disk will act strangely. Periodic VALIDATEing of the disk (type DC "V0") plus switching to a new empty disk to hold additional screens when an old one becomes 80-90% full will prevent any problems.

One other detail - the only time a Commodore disk drive knows that the floppy disk in it has been switched is by detection of a different disk ID#. When you NEW a disk, it is imperative you assign a unique 2-number or letter code for the ID# (this 2-digit code follows the disk name in the NEW command you sent the drive to format the disk). Never use the same 2-digit code for more than one disk. And it is a good habit, although not absolutely necessary, to initialize a disk that has just been swapped into the drive (DC "I0").

- ERROR MESSAGES

An error message is printed as a question mark ('?') followed by a message number or informative text. As long as the C64-FORTH definition named 'EMSGS' exists in the dictionary, the error messages printed will be fairly descriptive. Right after an error, executing WHERE will print out the line at fault and an arrow pointing to just after where the error occured. Look for missing spaces between FORTH words, looping and branching structures mispaired, values left on the stack will cause problems within looping and branching structures, etc. Sometimes just the message is printed as a warning (e.g. ISN'T UNIQUE or EMPTY STACK), and at other times control is returned to the FORTH input interpreter (as in USE ONLY WHEN LOADING). Screens 4 & 5 on the C64-FORTH disk contain copies of the error messages. In the glossary, refer to EMSGS and ERROR. If an error occured when defining a colon definition, and the definition cannot be FORGETten, execute SMUDGE. The most recent header can then be removed.

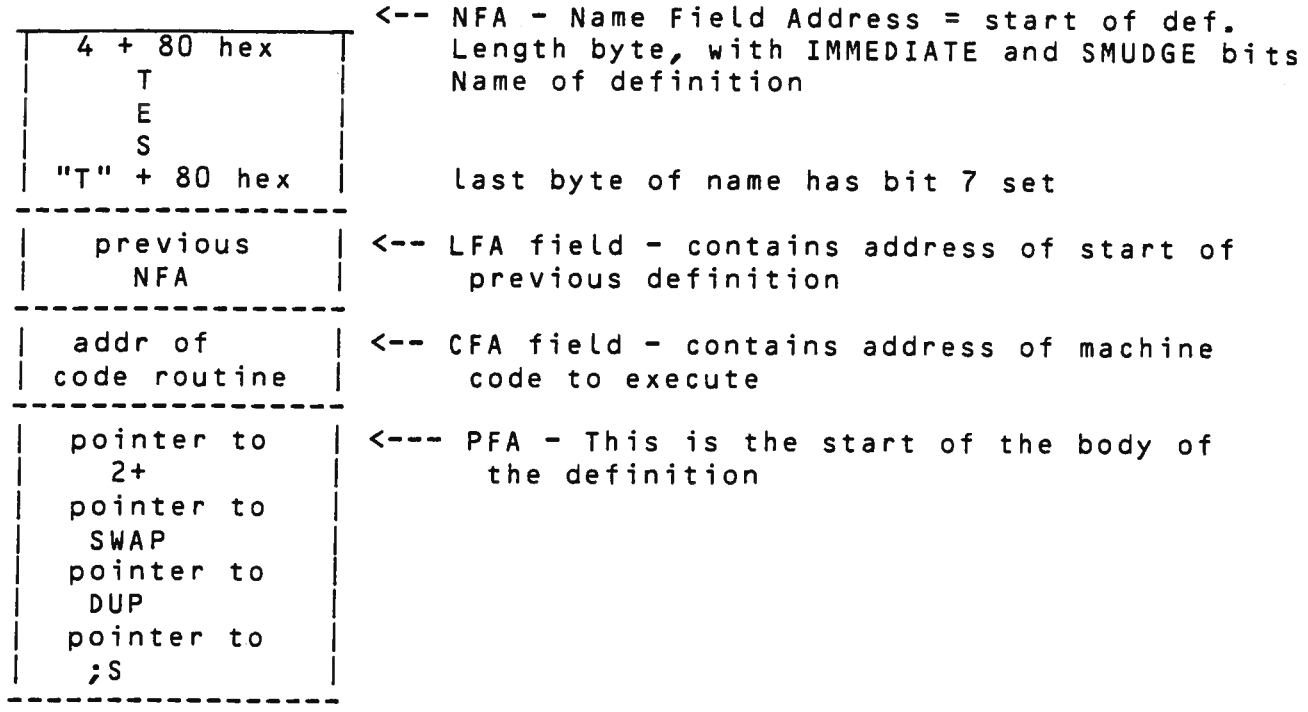
- USING IEEE INTERFACE CARTRIDGES

C64-FORTH will automatically work with any Commodore PET/CBM/VIC/C64 compatible disk drive. For the ones needing an IEEE interface, certain ones are available that are cartridge based and plug directly into the 64 cartridge slot. ROM code is a necessary part if any catridge-base interface. The internal 64 kernal I/O routines must be modified to divert commands and data from the serial bus to the IEEE interface. Since any cartridge ROM in the range 8000-9FFF is mapped out (made unaccessible as the RAM underneath is mapped in) when C64-FORTH maps out the BASIC ROMs, a way was provided to disable the disabling of the BASIC ROMs. Immediately after loading C64-FORTH, POKE 2067,255 before RUNning C64-FORTH. The BASIC ROMs, as well as any cartridge ROMs, will be left accessible. Only one thing to remember - there will be ROM code in the range 8000-BFFF once you enter C64-FORTH. Normally, the lowest disk buffer will overlap the BASIC ROM area. You must either decrease the number of buffers down to 2, or MEMTOP the system down to 8000 hex. MEMTOP is safer, since you won't have to worry about the end-of-the-dictionary overlapping the ROM code.

- THE STRUCTURE OF A DEFINITION IN MEMORY

We have been referring to different parts of a definition in memory. It is time to briefly cover the structure of a definition as it appears in memory. Lets show how the following definition will look in memory:

```
:  TEST  2+  SWAP  DUP  ;
```


4 + 80 hex
T
E
S
"T" + 80 hex
<-- NFA - Name Field Address = start of def.
Length byte, with IMMEDIATE and SMUDGE bits
Name of definition
last byte of name has bit 7 set
previous NFA
<-- LFA field - contains address of start of previous definition
addr of code routine
<-- CFA field - contains address of machine code to execute
pointer to 2+
pointer to SWAP
pointer to DUP
pointer to ;S
<--- PFA - This is the start of the body of the definition

The Name Field Address (NFA) is the address of the start of the header. Every definition works upwards in memory, so the NFA is the address of the lowest byte of that definition. The first byte of the header is used for various things. The lower 5 bits of it contain the count of the number of characters in the name. Bit 7 is always set to indicate the start of the header. Bit 6 is set to indicate if the definition is marked as IMMEDIATE, meaning it will execute immediately even within a new colon definition being created. Bit 5 is called the SMUDGE bit. When set, the definition will be listed with VLIST, but the definition cannot be executed, FORGOTten, etc. When : is used to create a new colon definition, the SMUDGE bit is set. Only when the terminating ; is excuted, does the SMUDGE bit get cleared. This prevents executing a colon definition that was not properly completed, due either to an error encountered within the source code of the colon definition, or the user just forgetting to terminate the definition with ; .

After the header byte, the name of the definition follows, with bit 7 being set in the last character of the name. After the name field comes the LINK field. This is a 16-bit pointer to the start of the previous definition (the NFA of the previous definition) WITHIN THE VOCABULARY IT WAS CREATED UNDER. This is what links all the definitions together for the purpose of vocabulary searches. The FORTH word LFA will provide the address of this field within this definition.

Next comes the CODE field. The code field contains a 16-bit pointer to a machine code routine. If this definition was defined as a colon definition, this will contain the address of a routine in FORTH that will sequence through the pointers contained in the body of this definition. If defined as a constant, the address will point to a routine that will push the 16-bit contents of the body onto the stack, and so on for other types of definitions. This Code Field Address points to this location.

And finally comes the body of the definition. The address of the first byte of this field is called the Parameter Field Address. If the definition is a colon definition, the body is made up of pointers to other word definitions. A colon definition terminates with a pointer to ;S (EXIT is the same thing). ;S tells the FORTH interpreter that the current definition is completed and to 'un-nest' one level - that is, return to the definition that called it.

If the definition is a variable or constant, then the body is only two bytes long, containing the current contents of the constant or variable.

- DEBUGGING FORTH ROUTINES USING TRACE FEATURE

A simple trace function is built into the system. It allows the user to single-step through the execution of a FORTH colon definition. Definitions composed of machine code cannot be stepped through. They will execute entirely in one step.

The trace feature is turned on by setting bits in location 132 (84 hex) of zero page memory. If bit 7 is set by writing 128 (80 hex) to this location, trace mode is immediately turned on. But since you have to go through a lot of FORTH kernal routines for the parsing and matching of input stream words, setting bit 6 is preferred for tracing one specific routine. There is a word in the FORTH dictionary which does that for you, called TON.

Example of usage:

```
3  7  TON  *
```

This will turn the trace mode on after the values 3 and 7 are pushed on the stack, and before * begins executing. You will skip over the FORTH system overhead, and tracing will begin just as the definition * is entered.

The name of the routine to be stepped through should follow immediately after turning on the trace; otherwise the end-of-the-line terminator, (the X <null> routine, which is actually defined as a word definition) will start tracing. Any parameters required by the routine to be examined should be put on the stack before trace is turned on.

Example:

```
( to trace the routine */ )
3  10  6  TON  */   ( turn on trace just before */ )
```

Two types of trace lines are shown. A normal line will look like this:

```
2031  */MOD  66  01F3  0006  000A  0003
```

All numbers are shown in HEX to provide a reasonable format on the 40-character C64 screen.

The first number displayed is the address within the current colon definition of the NEXT pointer to be fetched. The routine this pointer identifies will be the next to be executed, and the name of that routine follows the address.

The next number specifies that actual address of the bottom of the data stack (which is the TOP stack item). When the stack is empty, this value is 6C. It decrements by two for each data item on the stack (4 for each double number).

The return stack pointer follows. The return stack is used to hold the next pointer address when nesting down one level, DO loop index and limits, and temporary values placed on by >R. The return stack pointer also decrements by two for each entry placed on it.

On the remainder of the line, if any values are on the data stack, they will be displayed. Only the top four values are shown, with the top stack item being the leftmost.

Every time FORTH 'nests' down one level (starts executing a lower-level colon definition), TRACE will show this by printing a colon in column one of the display followed by the definition's name:

```
:  */MOD
```

'Unnesting' (returning to the higher-level definition) will not be shown, but can be recognized when ;S or EXIT is about to execute. After ;S or EXIT executes, the return stack pointer can be seen to increment by two showing the FORTH has unnested one level.

After each line is displayed, hitting just the RETURN key will single-step through the definition. If you wish to turn off trace mode and finish executing the definition at full speed, press R . To terminate execution of the routine, turn off trace mode, and return to the FORTH input prompt, press X .

There is one other feature of the trace mode. At the flashing cursor, you can press B as an option. B immediately executes a 6502 BRK (break) instruction. If you have a monitor wedge or cartridge installed, you will enter the monitor. Make note of the address printed upon entry to the monitor. You will need this address to return to C64-FORTH. You may now use the monitor to examine and modify memory locations, even to single-step (if it is a feature of your monitor of course!) through machine code routines if no FORTH system zero-page variables are disturbed. When you desire to return to FORTH, type G xxxx where xxxx is the address you wrote down when you entered the monitor. You will have executed the next FORTH word in the definition you were tracing, and you may continue to single-step through that definition. If the BRK vector has not been changed by a monitor prior to initial entry into C64-FORTH, then the FORTH redirects the BRK vector to a warm start routine, so the B command in trace does not crash when no monitor is present.

- MEMORY MAP

The COMMODORE-64 memory map, with C64-FORTH in control, looks like this (all addresses in HEX):

0000-00FF	6502 page zero memory. Contains data stack, C64 I/O and system variables.
0100-01FF	6502 return stack. Bottom 80 bytes used as FORTH terminal input buffer.
0200-03FF	C64 variables and buffers.
0400-07FF	Display memory.
0800-4xxx	C64-FORTH program. HERE leaves the address of the current end of the FORTH dictionary. PAD, the pointer to the text output buffer, always starts 68 bytes after the HERE address.
PAD+80 thru BD6F	Usable RAM for expanding the FORTH dictionary, machine code routines, work buffers, etc. Note that the C64-FORTH editor uses this area for cut-and-paste temporary storage.
BD70-CD7F	Block buffers. As shipped, C64-FORTH is setup for 4 buffers. If the number of buffers is changed, or MEMTOP is changed, then this address range will be different.
CD80-CDFF	User area. The user area is 128 bytes long, and is a collection of variables describing the characteristics of the current FORTH system. This address range will change if MEMTOP is changed.
CE00-CFFF	RS-232 buffers. If device #2 is OPENed, these buffers will be used by the 64 kernal ROM routines.
D000	By default, the highest address +1 of usable continuous RAM. MEMTOP may be used to lower the effective top-of-memory as perceived by the FORTH system.
D000-DFFF	COMMODORE-64 I/O register space.
E000-FFFF	COMMODORE-64 I/O and screen display ROMs. For high-resolution bit-mapped graphics, the RAM that 'hides' behind these ROMs may be used.

The COMMODORE-64 BASIC ROMs are normally in the address range A000-BFFF, but are mapped out by C64-FORTH. They are replaced by usable RAM. The ROMs are re-enabled when exiting to BASIC.

- SAVETURNKEY

SAVETURNKEY is the only way any version of C64-FORTH may be redistributed. It also has the advantage of making the application program more end-user oriented, since the FORTH system itself is no longer accessible. The program, after being loaded from BASIC and entered via RUN, initializes the system and immediately executes the top word in the FORTH vocabulary. Any error exits back into the FORTH kernal will restart that last definition. The RESTORE key also works properly and causes a restart.

The procedure for creating a turnkey system is as follows:

1) Write and debug your application program (enough said).

2) If desired, to shorten load time of the application program, you may forget the editor and error messages by:

   FORGET EDITOR

   You may also not need the floating-point extensions, which take up about 4K of space. FORGET FPOVER to recover that space too.

3) Load your application program onto the FORTH system.

4) Insert a pre-NEWed disk into the disk drive. Set SYSDEV# to the device # you wish the program to be saved to.

5) Type the word 'SAVETURNKEY' followed by the program name.

   Ex. SAVETURNKEY "SPACE WIDGETS"

6) SAVETURNKEY immediately asks whether you really do want to create a turnkey system, in case you were managing the clouds when you entered the command. Reply by pressing the 'Y' key if you weren't.

7) It responds by telling you to insert the next disk (pre-NEWed, of course), and to press any key. The program is saved on disk or tape, and the inquiry is repeated. Do this for as many disks or tapes as you wish to create. Since the FORTH system has been thoroughly corrupted, you'll have to reboot C64-FORTH when done.